Lazy Abstraction

by

Ranjit Jhala

B.Tech. Hons. (Indian Institute of Technology, New Delhi) 1999

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Thomas A. Henzinger, Chair
Professor George C. Necula
Professor Leo Harrington

Fall 2004

The dissertation of Ranjit Jhala is approved:

Chair

Date

Date

Date

University of California, Berkeley

Fall 2004

Lazy Abstraction

Copyright 2004

by

Ranjit Jhala

Abstract

Lazy Abstraction

by

Ranjit Jhala

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Thomas A. Henzinger, Chair

The enormous cost and ubiquity of software errors necessitates the need for techniques and tools that can precisely analyze large systems and prove that they meet given specifications, or if they don't, return counterexample behaviors showing how the system fails. Recent advances in model checking, decision procedures, program analysis and type systems, and a shift of focus to partial specifications common to several systems (e.g. memory safety and race freedom) have resulted in several practical verification methods. However, these methods are either precise or they are scalable, depending on whether they track the values of variables or only a fixed small set of dataflow facts (e.g. types), and are usually insufficient for precisely verifying large programs.

We describe a new technique called Lazy Abstraction (LA) which achieves both precision and scalability by localizing the use of precise information. LA automatically builds, explores and refines a single abstract model of the program in a way that different parts of the model exhibit different degrees of precision, namely just enough to verify the desired property. The algorithm automatically mines the information required by partitioning mechanical proofs of unsatisfiability of spurious counterexamples into Craig Interpolants. For multithreaded systems, we give a new technique based on analyzing the behavior of a single thread executing in a context which is an abstraction of the other (arbitrarily many) threads. We define novel context models and show how to automatically infer them and analyze the full system (thread + context) using LA.

LA is implemented in BLAST. We have run BLAST on Windows and Linux Device Drivers to verify API conformance properties, and have used it to find (or guarantee the

absence of) data races in multithreaded Networked Embedded Systems (NESC) applications. BLAST is able to prove the absence of races in several cases where earlier methods, which depend on lock-based synchronization, fail.

Acknowledgements

I will always be in the debt of my advisor, Tom Henzinger, for the support and guidance I have received over the last five years. He initiated me into the world of research by turning my attention towards various interesting problems including the one studied in this dissertation. He then listened, with unwavering enthusiasm and patience, to all all my ill-conceived and nebulous ideas and deftly steered me towards solutions. I can only hope that in addition to his brilliance, some of his taste in research questions, his ability to crystallize a tangled mess of ideas into a precise solution, and his unshakeable calm has rubbed off on me over the last five years.

I am absurdly lucky to also have had the chance to work with and see first hand the ingenuity of Ken McMillan. Ken's insights into various problems and techniques made every conversation with him an exhilarating learning experience, even though they left me mentally exhausted from trying, in vain, to keep up! His work has been the inspiration for many of the ideas in this dissertation, and will, I expect, continue to provide fuel for my investigations over the next few years. When I grow up, I want to be like Tom and Ken.

This work crosses the traditional boundaries drawn between the areas of Model Checking, and Deductive methods for program verification. Most of the credit for this goes to George Necula from whom I have learnt what I know of Programming Languages and Automated Deduction. George, together with Alex Aiken and Ras Bodik, has been a font of helpful advice about various aspects of life within and without academia. I am grateful to Leo Harrington for teaching me about Mathematical Logic, and surrendering some of his time to serve on my committee. While I took his class several years ago, I find results from it popping up everywhere in my present work.

I could not have asked for a better partner in crime than Rupak Majumdar, who must get a lion's share of the credit for this work. On countless occasions we have worked through solutions by bouncing ideas off each other and by solving questions that the other raised but was unable to answer. Working in a team with him was, without question, the best part about graduate school for me, which, if you went to grad school at Berkeley, is saying a lot. I also thank Gregoire Sutre for stoically working with us even as we trampled

upon all his sensibilities when building BLAST , leaving him to clean up the code after us. Luca de Alfaro and Shaz Qadeer have also, by way of discussions on matters technical and otherwise, greatly contributed to my maturing as a researcher. Tom Ball and Sriram Rajamani's pioneering work on SLAM led to this dissertation – I am very grateful to them for generously sharing their ideas and insights with me.

None of this work would have been possible if I hadn't been immersed in an community peopled by an unparalleled group of students – both within Tom's group as well as in the larger OSQ group. I am especially grateful to Wes Weimer, not only for the delicious flank steak and stuffed mushroom dinners, but also for his contructive feedback and advice that greatly informs much of this work, and the patience with which he taught me about the inner workings of various tools. Thanks are also due to Wes and Rupak for serving as supremely informed oracles about various research areas about which I, to my undying shame, found myself completely ignorant.

Living in Berkeley over the last five years has been one of the best things to have happened to me; the combination of the glorious weather and cosy cafes (especially Brewed Awakenings and Caffe Strada, where much of this work was done) has been a great source of motivation. The greatest appeal of Berkeley lies in the friends I've made here, of which there are far too many to name! I shall single out my two roommates Anupam Gupta and Kunal Talwar for putting up most sportingly with the mess that surrounds me, Iordanis Kerenidis, Kevin Chen, for participating in frivolous debates that (doubtless) kept my faculties sharp, Hoeteck Wee, Andrej Bogdanov, Sam Riesenfeld and Ashwin Nayak for participating in various culinary adventures and Chris Harrelson and Sara Rahimian for putting up (quite literally) with my foibles and tutoring me about the finer points of Americana.

I have been lucky to have been blessed with parents who have suffered my every whim without complaint, have been an endless source of love, support, and (mostly) good advice. One day, I hope to be able to tell them exactly what it is that I do. Finally, to Kamalika. I don't know how to thank her enough. She makes it all seem worthwhile.

For Mr. and Mrs. Zed...

Life can only be understood going backwards,
but it must be lived going forwards.

Contents

1 Introduction **1**

2 Programs and Abstractions **15**

2.1 Labeled Transition Systems 15

2.1.1 Symbolic Region Structures 16

2.1.2 Symbolic Abstraction Structures 17

2.1.3 Predicate Abstraction 18

2.2 Imperative Programs 19

2.2.1 From Imperative Programs to LTSs 19

2.2.2 Predicate Abstraction for Imperative Programs 23

2.3 The Safety Verification Problem 28

3 Lazy Abstraction **31**

3.1 A Locking Example 35

3.1.1 Verification 36

3.2 Symbolic Reachability with Refinement 42

3.2.1 Reachability with refinement 43

3.2.2 Counterexample-driven refinement 46

3.3 A Refine operator for Imperative Programs 51

3.3.1 Overview 52

3.3.2 Interpolants from Proofs 54

3.3.3 The Algorithm Refine 56

3.4 Theoretical Issues 64

3.4.1 Termination 64

3.4.2 Finite predicate abstraction is undecidable 65
3.5 Related work 67

4 Applications 69
4.1 Device Driver Verification 71
4.2 Temporal-safety proofs from Reachability Trees 75
4.2.1 Overview 76
4.2.2 Verification Conditions 78
4.2.3 VCs and Proofs via Lazy Abstraction 79
4.2.4 Experiments 83
4.3 Tests from Counterexample Traces 84
4.3.1 Overview 86
4.3.2 Testing Framework 90
4.3.3 Test Suite Generation 92
4.3.4 Experiments 97

5 Multithreaded Programs: Context Inference 100
5.1 An Example 103
5.1.1 Threads 104
5.1.2 Thread-Context Programs 105
5.1.3 Verification by Abstraction 106
5.1.4 The Algorithm CIRC 108
5.2 Safety Verification of Multithreaded Programs 112
5.2.1 Multithreaded Labeled Transition Systems 113
5.2.2 Thread-Context Verification 113
5.3 Abstractions 115
5.3.1 Main Thread: Data Abstraction 116
5.3.2 Environment Thread: Control Abstraction 117
5.3.3 Context: Counter Abstraction 119
5.3.4 Abstracting Thread-Context Programs 119
5.4 Verification by Thread-Context Abstraction-Refinement 120

5.4.1 Checking 121
5.4.2 Inference 125
5.5 Race Detection for Multithreaded Imperative Programs 128
5.5.1 MLTSs from Imperative Programs 128
5.5.2 Predicate Abstraction 129
5.5.3 The Race Detection Problem 129
5.5.4 Procedure Refine 130
5.6 Experiences 131
5.7 Completeness of Counter Abstractions 134
5.8 Related Work 139

Chapter 1

Introduction

This dissertation proposes new methods for the *Safety Verification problem*, which is, given a program, an *initial* state[1] from which the program begins execution, and a set of *error states*, to decide whether there is an execution of the program that leads it from the initial state to an error state.

A brief history of Program Verification

The Safety Verification problem can be said to have been born at the same time as Computer Science itself, in Turing's work on the Halting problem[Tur36]. Therefore, that paper was, in a way, the death of the problem, since it was shown that the task is theoretically unmechanizable. However, just as mathematicians did not stop proving theorems as a result of Godel's theorem, leading computer scientists in the last four decades, remained undaunted, and recognizing verification to be a fundamental problem of their subject, made significant advances by developing methods of showing that a program met its specifications.

Safety and Liveness Verification are the two main problems in the area of Program Verification, which, as the name suggests, is concerned with guaranteeing that programs had certain desirable properties (*i.e.,* they met certain specifications). The former deals with *safety* properties which stipulate that the program never performs an undesirable operation (*e.g.* never divides by zero). The latter pertains to *liveness* properties which stipulate that the program eventually performs desirable operations (*e.g.* eventually terminates).

The history of Program Verification dates back to the mid-sixties, when several of

[1] Configuration, *i.e.,* values of all variables, initial program counter, stack

the founding fathers of Computer Science such as Robert Floyd, Tony Hoare and Edsger Dijkstra realized that one of the most important challenges that Computing faced was the need to tame complexity by way of devising ways of devising methodologies that enabled engineers to build systems such that rigorous claims could be made about their properties.

One way to study the properties of a program is to treat it as a black-box and execute it. While this approach has many merits, foremost amongst them being ease of implementation, its chief drawback is that it only reveals how the program behaves for a particular input value, and reveals little about the outcome of executing the program on a different input. In order to guarantee that some property held regardless of the input, one would have to execute the program over *every* possible input, which was clearly undesirable.

Hence, the most basic requirement was to be able to open up the black-box, and devise ways of being able to characterize the behaviour of the program given its text. This led to the development of higher-level programming languages with precise semantics, using which programs could be treated as mathematical objects, thus allowing calculational methods to be applied in order to reason about their properties. Once the foundations for precisely describing program semantics[Win93] were laid down, researchers began to follow two distinct but overlapping paths.

The first was designing programming languages of increasing sophistication, which promoted various styles of programming while subtly restricting the programmer's ability to shoot herself in the foot. The second was the development of techniques of reasoning about the properties of programs, in a manner that was largely independent of the underlying language. These techniques were both methodological and algorithmic, and in many cases started off as the former, but ended in the latter category.

Over the years the techniques organized themselves into distinct sub-areas, namely Type Systems, Flow-based Analysis, Theorem Proving and Model Checking, each with its own community, philosophy, conferences and jargon. It is only over the last few years that lines of communication have opened up between these areas. Researchers have realized that by taking a few steps back, one can view enormous similarities between the sub-areas, and more importantly, exploit their complementary strengths to devise more powerful and efficient analyses.

The simple insight that unifies all these sub-areas is that to prove that the program never enters an undesirable state, one has to effectively compute from the program's description, the set of possible states that the program may ever enter. As this set is not computable in general, what needs to be done is to demonstrate (by construction) the existence of a safe set that contains all the states the program may enter, but does not contain an undesirable(unsafe) state. From this point on, the sub-areas diverge, owing mainly to differences in the kinds of error states and the programs being analyzed. While each has a rich literature of its own, it is important to remember that at their core sub-area approach is attempting to effectively demonstrate the existence of this intermediate set. To do so, each sub-area begins with a language for describing such sets, armed with which researchers begin to describe ways to compute the set.

1. Type Systems.

A program manipulates data of different *types*,*e.g.* integers, strings, lists, functions *etc.*. Perhaps the most basic correctness requirement of a program is that it only performs operations on data for which the operation is valid. For example, addition should only be performed on two integers, only strings or lists can be concatenated, only an integer can have value greater than zero and so on. In this setting, the undesirable states are those where a type-error occurs. To ensure that such a state is not reached, instead of finding the the exact set of states that the program may be in, the methods focus on the only relevant aspect about the program's states, namely the *type* of the value of the variable at each point during the execution. The problem of then computing the safe set reduces to *inferring* the type of the run-time values of the program's variables. The description of the safe set is a type assignment for the program's variables, and this leads to an algorithm to infer the types by setting up a system of constraints the solution of which is an appropriate type assignment. A program is *type safe* iff the type assignment found for it corresponds to a safe set, *i.e.,* one not containing error states. The precision with which the safe set can be described depends on how precisely one can state the type of a variable *i.e.,* the precision of the type system.

The groundbreaking work in this area was Milner's invention of the programming language ML [Mil78], and the subsequent type inference algorithm for ML[DM82]. Taking the

broader view that the type of a value is simply a predicate describing the value (and hence corresponding to a subset of values), has led to more sophisticated type systems capable of expressing richer properties about programs. Two examples are Linear Types[Wad90], which allow one to stipulate that an object has a unique pointer to it, and the notion of Typestates[SY86; FD04], using which one can express properties that an object may exist in different configurations or typestates at different points in the program, and certain operations change the configurations. Other interesting applications of this generalized view of types are found in [Eva96] where the author uses types to capture whether a pointer is `NULL` or not, and [OJ97] where the authors cleverly adapt the ML type inference algorithm to "reconstruct" the most general types for variables in C programs, which among other things, enables a user to know when the data corresponding to two variables with the same C type (*e.g.* integers) can have different representations, (*e.g.* one integer variable actually corresponds to a file handle while another corresponds to a socket). A generalization of this approach is [FFA99] where a user can add arbitrary predicates or *qualifiers* on top of the C type system, and specify properties of the qualifiers. This method has been used to find subtle security flaws in Linux code, where a variable whose value may have been controlled by a malicious adversary was used to perform a critical operation [USW01; JW04].

2. Flow-based Analysis. A class of program properties that is especially useful when compiling code pertains to what data values are available at any point in the program. A expression may be computed at one point and its constituent variables not modified for several instructions. If that expression is needed again, it need not be recomputed. More broadly, one can view the expression as being a fact *generated* at the point of first evaluation, *killed* at any point when a constituent variable is modified and otherwise flowing untouched across the operation. Hence, one can view the program as a graph whose vertices correspond to operations and edges exist between successive operations. For each operation (vertex), we wish to compute the set of facts that hold when before the program executes the operation. For each operation, the semantics of the program describe which facts are killed by an operation and which are generated by the operation, and hence the facts that hold after an operation are those that held before minus those that were killed plus those

that were generated. The facts that hold before an operation with multiple predecessor is specified by the *combination* of the facts that held after the individual predecessors. In a seminal paper [Kil73], Kildall showed how a variety of hitherto ad-hoc program analyses could be phrased be describing 1) the set of dataflow facts, 2) how a fact was generated or killed by an operation and 3) how to combine facts. Once the above were supplied, Kildall showed how computing the set of facts that held at each point reduced to solving a system of constraints called data flow equations. In another highly influential paper [CC77], Cousot and Cousot showed how Kildall's approach could be connected rigorously with the semantics of the program by demonstrating that the dataflow facts at each point corresponded to states the program could be in at that point, called the *concretization* of the dataflow facts. Thus, instead of viewing operations as generating or killing facts, one could view them as *transferring* the set of facts that held upon input to some set of facts that held upon output. Two other significant papers in this area were [SP81; RHS95] which showed how to perform the above analyses on programs with procedures via a technique called *summarization*, namely the caching of the analysis done at one call site when analyzing another.

Over the last few years researchers have begun applying these methods for verification. The safe set computed by these analyses is expressed via the dataflow facts that hold at each program point. Thus, to get an appropriate safe set, one must choose an appropriate set of dataflow facts. A rich set of facts leads to a more precise safe set, but comes at a higher computational cost than a smaller set of facts. Some examples are [DC94; ECCH00; DLS02], where there is a finite set of facts that correspond directly to the property to be verified, and [LAS00; BCC$^+$02], where additional facts which are effectively rich predicates describing program states are used in order to gain greater precision.

3. Theorem Proving. The most general techniques for Program Verification are those classified as Theorem Proving or Deductive Methods. Here, the language used to describe the program states are first-order formulas over program variables. Hence, one can prove properties where the undesirable or error set of states can be described using such formulas called the error condition. Floyd [Flo67] and Hoare[Hoa69] showed how to describe the semantics of programs using such formulas. Given a set of states as a formula, they showed

how to compute formula corresponding to the set that results from an individual operation using predicate transformers. Hence, to show that an arbitrary property, expressible in the logic, held at a program point, one had to compute the safe set for that program point, namely a formula that described an overapproximation of the states the program could be in at that point. The difficulty arises when the program has loops as iterating the predicate transformer around the loop may not terminate. Hence, one must manually supply a *loop invariant*, a formula describing all the states the program could be in regardless of the number of times it went around the loop. Once such invariants are supplied for every loop, one can mechanically compute formulas describing (overapproximations of) the set of states the program can ever be in at every point. Checking that the set was safe reduces to checking that the formula implies the negation of the error condition. In his dissertation [Nel81], Nelson describes efficient decision procedures that can mechanically prove the validity of such implications for formulas that straddle several interesting theories.

Since then, owing to the high cost of the technique, it has been used in situations where the system is small but the correctness criterion is very complex, *e.g.* the verification of microprocessors (where the microprocessor is described as a LISP program)[KMM00] and numerous other examples of hardware systems. There are systems like ESC [FLL+02] which allow users to supply loop invariants and uses them to verify user defined assertions. Other tools such as Prefix[BPS00] use theorem proving technology for bug-hunting. Necula [Nec97a] showed how a formal proof of the above implication could be used as a certificate that the program satisfied some property and hence was safe to execute.

4. Model Checking. Every program, at its lowest level, can be described as a set of states and a binary transition relation. The set of states corresponds to the states the program can be in, and two states are in the transition relation if the first can transition to the second in one step during the execution of the program. Equivalently, one can view the program as a directed graph with vertices representing states, and directed edges between states in the transition relation. Checking if a error state cannot be reached can be done by enumerating the states reachable from the initial state, which is feasible if the graph is finite. The papers of Clarke and Emerson [CE81] and Queille and Sifakis[QS81], showed how to algorithmically decide if programs represented via such finite graphs satisfied a richer class of *temporal logic*

specifications like those of Manna [Man69] and Pnueli [Pnu77], which stipulate restrictions on the *order* in which various states are visited. These algorithms worked on this graph representation and essentially enumerated the states of the program. Hence, they worked only for relatively small finite state systems. In his dissertation, McMillan [McM93] showed how to expand the scope of those algorithms by using propositional formulas to represent sets of states, and using Binary Decision Diagrams [Bry86] to efficiently manipulate the boolean formulas, in a technique called "Symbolic Model Checking". Following that model checking made significant inroads in hardware verification, as the complex concurrent nature of circuits, the high cost of bugs and their boolean nature made them perfect for this approach.

While software is not finite-state, several software model checkers[God97; HP00; MPC+02; Hol00; CDH+00] have been used used as systematic testing engines that explore the states of the system until they run out of memory (or time). Such have been used to find subtle errors in several complex systems.

Recent years have seen a marked increase in activity in the area of Program Verification owing to two reasons. First, increasing dependence on software systems has ensured that their reliability is no longer a luxury but an urgent neccessity. Additionally, the increasing size of programs renders traditional methods like code inspection and testing increasingly insufficient, as both methods are swamped by the proliferation of "corner cases". This, coupled with the impossibility of specifying what it means for an large software system, *e.g.* an operating system, is "correct", motivates the neccessity for, to quote Tony Hoare, "proving little theorems about big programs", *i.e.,* proving that large software systems meet a set of partial specifications, the proof serving to give some confidence about the reliability of the system. For example, instead of showing a device driver behaves correctly (indeed it is not even clear how to precisely and completely specify what a device driver should do in a manner significantly smaller than the driver itself), we wish to prove that it calls certain OS functions in a certain order, or that it relinquishes shared resources after acquiring them.

Second, the toil of several decades, and ingenuity of researchers working in the area, on

the shoulders of which our work stands, has led to the discovery of powerful and efficient techniques for program analysis and verification. These, combined with the raw computing horsepower available today, have enabled researchers to devise analyses for large programs.

Counterexample-Guided Abstraction-Refinement

For verification, the analysis of the program must be *precise* and *scalable*. Precision is required so that the analysis is neither fooled by spurious errors nor overlooks genuine errors. Scalability is a must so that the method works for large software systems where the need for analysis is most acute. The trouble is that these two features are often mutually exclusive: flow based analyses [FTA02; DLS02] achieve scalability by fixing a small domain of dataflow facts to be tracked, and compute flow functions over the abstract semantics of the program on this fixed set. For complicated properties, the set of facts that are tracked is too small, leading to a high rate of false positives,*i.e.,* a large number of the bugs reported, turn out to be behaviors that never arise when the program executes. Model checking based approaches on the other hand while precise and path-sensitive, often end up tracking too many facts, so state explosion comes in the way of scalability.

To avoid the pitfalls arising from from using a fixed set of facts, much recent interest has focused on analyses that automatically tune the precision of the analysis using false positives *i.e.,* in a *counterexample-guided* manner using the following loop [AIKY95; BR01; CGJ+00; Sai00]:

Step 1 ("abstraction") A finite set of predicates is chosen, and an abstract model of the given program is built automatically as a finite or push-down automaton whose states represent truth assignments for the chosen predicates.

Step 2 ("verification") The abstract model is checked automatically for the desired property. If the abstract model is error-free, then so is the original program (**return** "program correct"); otherwise, an abstract counterexample is produced automatically which demonstrates how the model violates the property.

Step 3 ("counterexample-driven refinement") It is checked automatically if the abstract counterexample corresponds to a concrete counterexample in the original program. If

so, then a program error has been found (**return** "program incorrect"); otherwise, the chosen set of predicates does not contain enough information for proving program correctness and new predicates must be added. The selection of such predicates is automated, or at least guided, by the failure to concretize the abstract counterexample [CGJ+00].

Goto Step 1.

In our thesis we will study ways to scale this basic paradigm to obtain techniques for solving the safety verification problem for large programs. We first propose a new method for the analysis of sequential programs, and then show how to extend it to multithreaded programs, which are the composition of several, possibly unboundedly many, sequential programs running concurrently and communicating via shared variables.

Sequential Programs

The main problem with the approach outlined above is that both Step 1 and Step 2 are computationally hard problems, and without additional optimizations, the method does not scale because of the following reasons:

1. Different parts of a program's state space are safe for different reasons. A monolithic abstraction must contain *all* the predicates for the different parts of the state space and thus is too detailed as the number of predicates grows with the size of the program text and state explosion gets in the way of scalability.

2. The method constructs an abstraction of the entire state space which is very expensive (more so when a monolithic set of predicates is used) since one must know for every possible abstract state and operation all the successor abstract states. Typically the set of reachable abstract states is very sparse and so building the transition relation on-the-fly offers considerable savings.

3. A coarser abstraction may suffice to show a large part of the state space is safe, yet upon finding a spurious counterexample, the method outlined above, fails to exploit this fact and repeats the work of exploring that part of the state space.

Hence, for scalability our analysis must: (1) localize the abstraction – *i.e.,* instead of a single monolithic abstraction we must be able to partition the state space appropriately and use different sets of predicates everywhere, (2) restrict itself to the small set of reachable states and (3) avoid re-exploring safe parts of the state space. We first present an algorithm called Lazy Abstraction which achieves the above goals.

Intuitively, lazy abstraction proceeds as follows. In Step 3, call the abstract state in which the abstract counterexample fails to have a concrete counterpart, the *pivot state.* The pivot state suggests which predicates should be used to refine the abstract model. However, instead of building an entire new abstract model, we refine the current abstract model "from the pivot state on." Since the abstract model may contain loops, such *refinement on demand* may, of course, refine parts of the abstract model that have already been constructed, but it will do so only if necessary; that is, if the desired property can be verified without revisiting some parts of the abstract model, then our algorithm succeeds in doing so. The algorithm integrates all three steps by constructing and verifying and refining *on-the-fly* an abstract model of the program, until either the desired property is established or a concrete counterexample is found. Upon termination with the outcome "program correct," the proof is not an abstract model on a global set of predicates, but an abstract model whose predicates change from state to state.

Lazy abstraction adds demand-driven path sensitivity to traditional dataflow analysis of programs. It is *sound*, in that if the algorithm reports that a program satisfies a safety property, then that is in fact the case. It goes beyond traditional dataflow analysis, as the counterexample refinement phase rules out false positives. In case an error is found, the model checker also provides a counterexample trace in the program showing how the property is violated. A key problem that we solve is how to use false positives to refine the abstraction – this requires that we learn not just a (small) set of predicates that rule out the counterexample, but also *where* we must track those predicates.

We solve both problems using the following observation: the *reason* why a trace is infeasible is succinctly encoded in a *proof* that the trace is infeasible, and so the appropriate abstraction can be culled from the proof. The difficulty in extracting the facts from the proof is that the proof uses the entire history of the trace, while our analysis, and hence

our facts must be over "current" relationships between variables at the various time slices in the trace. We introduce a method by which the proof can be sliced appropriately, to yield the relevant facts at each point in the execution of the trace. First, given a spurious trace, we build a *trace formula* (TF), which is the conjunction of several constraints, one per instruction in the infeasible trace, such that the TF is is satisfiable iff the trace is feasible. We then use Craig's interpolation theorem [Cra57] to extract, for each point in the trace, the relevant facts from the proof of unsatisfiability of the TF. Given two formulas φ^-, φ^+, whose conjunction is unsatisfiable, the Craig interpolant of (φ^-, φ^+) is a formula ψ, such that: (i) $\varphi^- \Rightarrow \psi$, (ii) $\psi \wedge \varphi^+$ is unsatisfiable, and (iii) ψ is made up of the symbols common to φ^- and φ^+. If φ^+ is the part of the TF that represents a prefix of an infeasible trace, and φ^- encodes the remainder of the trace, then the Craig interpolant ψ represents precisely the facts, as relations between current values of the variables, which need to be known at the cut-point of the trace in order to prove infeasibility. This work is inspired by [McM03] where it is shown how interpolation can be used for image computation thus yielding a totally SAT based approach for model checking systems with only boolean variables.

BLAST

We have implemented the above algorithms in a software verification tool called Blast , the Berkeley Lazy Abstraction Software Verification Tool, available at `http://www.eecs.berkeley.edu/~blas`
We report on several applications of Blast .

1. Verification. We have applied Blast to verify complex temporal safety properties of device drivers. Device drivers are written at a fairly low level, but must meet high-level specifications, such as locking disciplines, which are difficult to verify without path-sensitive analysis. They are critical for the correct functioning of modern computer systems, but are written by untrusted third-party vendors. Some studies show that device drivers typically contain 7 times as many bugs as the rest of the OS code [CYC+01]. Using Blast, we have run 10 examples of Linux and Windows device drivers, of up to 60K lines of C code. We have been able to discover several errors, as well as show that some drivers satisfy the given specifications. Owing to the fact that Blast only looks at the reachable states, avoids repeating work and tracks just a few predicates at every point, we have been able

to precisely model check programs considerably larger than have been reported before, including a driver which has 138,000 lines of C code, after pre-processing, *i.e.,* including kernel stubs, libraries etc.[2] We found several behaviors that violate the specification. Even though 382 predicates are required in all to show correctness, at each program point, the number of interesting predicates is on average at most 8.

2. Certification. Traditionally, in the case that a program met the specification, the user has had to take the tool's word. The size and complexity of the verification engine, makes the *trusted computing base* much larger than desirable, and hence, previous approaches fail to meet the goal of enabling the construction of *trusted* software. Trust is especially important for low-level systems code, which usually cannot be shielded from causing mischief by runtime protection mechanisms. Trust requires technologies for the *certification* of software, which assure users that the programs meet their specifications, *e.g.* that the code will not crash, or leak vital secrets. Like verification, certification is most effective when performed for actual code, not for separately constructed abstract models. *Proof-carrying code*(PCC) [Nec97b] has been proposed as a mechanism for witnessing the correct behaviour of untrusted code. Here, the code producer sends, along with the code, a proof that the code adheres to some safety policy. The code consumer has merely to run a small and trustable proof checker to check that the proof is consistent with the program. We show how to extract from BLAST 's data structures, a machine checkable *proof of correctness* in the case when the checker reports the system to be safe, and thus for the first time, show how to automatically construct certificates of correctness for complex temporal safety specifications.

3. Test Generation. The most popular approach to analyzing programs is to *execute* them on a set of test inputs. The biggest problem with this approach is constructing test vectors with desirable properties, *i.e.,* that cause the program to execute in certain ways. For example, the programmer may wish to devise a test that causes the function f to be called with the first formal parameter being -1. We show how to use BLAST to generate a test vectors that cause the program to execute thus, by treating the negation of the condition as a safety property, running BLAST to find a counterexample behaviour

[2] Just the driver is typically a factor of 4–5 less than the preprocessed size.

of the program *i.e.,* one where f is called with parameter -1, and then extracting a test vector from the counterexample. Hence, via BLAST 's static analysis, we can automatically construct *designer test suites* which can then be used for various dynamic analyses, instead of the random tests that have traditionally been used.

Concurrent Programs

The traditional strength of model checking lies in the analysis of concurrent systems, such as multithreaded programs, in which errors are notoriously difficult to reproduce. Concurrency, however, is a major practical obstacle to model checking: the interleaving of concurrent threads causes an exponential explosion of the control state, and if threads can be dynamically created, the number of control states is unbounded.

One approach [Jon83] is to consider the system as comprising a "main" thread and a *context* which is an abstraction of all the other "environment" threads in the system, and then verifying (a) that this composed system is safe ("assume"), and, (b) that the context is indeed a *sound* abstraction ("guarantee"). Once the appropriate context has been divined, the above checks can be discharged by existing methods [God97; CDH+00; Hol00; HP00; FQS02]. Additionally, the remaining data abstraction can be performed automatically using counterexamples [BR00; HJMS02; COYC03]. Note that either check may fail due to imprecision in the context, leaving us with no information about whether the system is safe or not.

Consequently, the main issues are: (a) what is a model for the *context* that is simultaneously (i) abstract enough to permit efficient checking and (ii) precise enough to preclude false positives as well as yield real error traces when the checks fail, and (b) how can we infer such a context automatically. It turns out, that for many multithreaded programs of interest, the context must allow the environment threads to have private state, which leaves us with the problem of tracking the private state of possibly arbitrarily many environment threads.

We give a novel way to construct stateful contexts, by representing individual environment threads as abstract finite state machines, and tracking arbitrarily many threads by *counting* the number of threads at each abstract state. We present a new way to in-

fer stateful contexts, by embedding the abstract reachability analysis used for sequential programs, inside an outer loop that iteratively constructs an appropriate context, by using the reachability information computed by the inner loop, using spurious counterexamples as before, to refine the abstraction of the system.

To demonstrate the practicality of the method, we have implemented this algorithm, called CIRC, in our C model checker BLAST . We ran CIRC to look for *race conditions* on several networked embedded systems applications written in NESC [GLvB+03], which use non-trivial synchronization idioms, that cause previous, imprecise analyses to race false alarms. We were able to find potential races in some cases and prove the absence of races in others.

Organization

In Chapter 2 we formally define programs, abstractions and the safety verification problem. In Chapter 3 we present the Lazy Abstraction algorithm for sequential programs, and in Chapter 4 we discuss the implementation in BLAST and several applications of BLAST , namely driver verification, constructing proofs of correctness, and generating test inputs. In Chapter 5 we present the generalization of Lazy Abstraction to the verification of multithreaded programs using Thread-Context Reasoning.

Bibliography

Chapter 2 and parts of chapter 3 is based on work that was first presented at the *29th Annual ACM Symposium on the Principles of Programming Languages,* 2002 [HJMS02]. The remainder of chapter 3 first appeared in a paper presented at the *31st Annual ACM Symposium on the Principles of Programming Languages,* 2004 [HJMM04], which also contains a part of chapter 4. The work on certification, in chapter 3, appeared at the *15th International Conference on Computer-Aided Verification,* 2002 [HJM+02], and the work on test generation was presented at the *Xth ACM/IEEE International Conference on Software Engineering,* 2004 [BCH+04b]. The work in Chapter 5 was presented at the *2004 ACM Symposium on Programming Language Design and Implementation,* [HJM04] and builds upon ideas presented at the *16th International Conference on Computer-Aided Verification,* 2003

[HJMQ03] .

Chapter 2

Programs and Abstractions

We begin by formally defining our model of programs, and defining the structures that we shall use to analyze the semantics of the programs.

We use *italics* for predicates, variables in our algorithms, `typewriter` font for code, and serif for functions, maps, and algorithms that we define. We use maps quite heavily: for a map F, we denote function application as $F.x$ where x is the argument for F, we assume "." is left associative, *i.e.,* $F.x.y$ is $(F.x).y$, and we denote $F[a \mapsto b]$ for the new map which equals F everywhere except at a where its value is b, *i.e.,* $F[a \mapsto b]$ is an abbreviation for the map $(\lambda x.$ if $x = a$ then b else $M.x)$.

2.1 Labeled Transition Systems

For a set X of variables, $\mathsf{V}.X$ denotes the set of valuations to X. We call the set $\mathsf{V}.X$, the set of X*-states*. A *labeled transition system* (LTS) $\mathcal{S} = (X, \Sigma, \rightsquigarrow, S_0)$ consists of:

1. A set of variables X,
2. A set Σ of *labels*,
3. A *labeled transition relation*, $\rightsquigarrow \subseteq \mathsf{V}.X \times \Sigma \times \mathsf{V}.X$; a transition $(s, l, s') \in \rightsquigarrow$ is written $s \overset{l}{\rightsquigarrow} s'$, and,
4. A set of *initial* states $S_0 \subseteq \mathsf{V}.X$.

We call Σ^*, *i.e.,* finite sequences of elements of Σ, the set of Σ*-traces*. For a trace σ, we denote by $\sigma.i$ the ith label of the trace. The relation $\rightsquigarrow$ is extended to traces as follows: $s \overset{\epsilon}{\rightsquigarrow} s'$ iff $s = s'$, and $s \overset{l\sigma}{\rightsquigarrow} s'$ iff there exists a state s'' such that $s \overset{l}{\rightsquigarrow} s''$ and $s'' \overset{\sigma}{\rightsquigarrow} s'$.

2.1.1 Symbolic Region Structures

Symbolic algorithms[McM93] on labeled transition systems manipulate regions, where each region represents a set of states. Following the framework of symbolic transition systems [FIS00; HM00], we define a (*symbolic*) *region structure* $(R, \bot, \sqcup, \sqcap, \mathsf{post}, [\![\cdot]\!])$ for the labeled transition system $\mathcal{S}$ to consist of a set R of *regions*, an element $\bot$ of R, two total functions $\sqcup, \sqcap$: $R \times R \to R$, one function post: $R \to \Sigma \to R$, and a total *extension* function $[\![\cdot]\!]$: $R \to 2^{\mathsf{V}.X}$, such that for all regions $r, r' \in R$ and every label $l \in \Sigma$, we have:

$$[\![\bot]\!] = \emptyset \tag{2.1}$$

$$[\![r \sqcup r']\!] = [\![r]\!] \cup [\![r']\!] \tag{2.2}$$

$$[\![r \sqcap r']\!] = [\![r]\!] \cap [\![r']\!] \tag{2.3}$$

$$[\![\mathsf{post}.r.l]\!] = \{s' \mid \exists\ s \in [\![r]\!].\ s \overset{l}{\rightsquigarrow} s'\} \tag{2.4}$$

The intention is that the region r represents the set $[\![r]\!]$ of states. A region structure carries with it a natural preorder (*i.e.,* a reflexive and transitive relation) $\sqsubseteq$ defined by $r \sqsubseteq r'$ if $[\![r]\!] \subseteq [\![r']\!]$, and a natural equivalence $\equiv$ defined by $r \equiv r'$ iff $[\![r]\!] = [\![r']\!]$. The region structure is *computable* if the functions $\sqcup$, $\sqcap$, post, and $\sqsubseteq$ are computable. The function post is extended to traces as: $\mathsf{post}.r.\epsilon = r$ and $\mathsf{post}.r.l\sigma = \mathsf{post}.(\mathsf{post}.r.l).\sigma$.

Example 1 The lattices used in dataflow analysis [ASU86] defines a region structure with R corresponding to the carrier set, $\bot$ the bottom element of the lattice, $\sqcup$ and $\sqcap$ the join and meet of the lattice respectively, and $[\![r]\!]$ the set of program states mapped to an element in the lattice. Region structures for models of computation such as counter automata (resp. FIFO automata, timed automata) can be designed based on Presburger formulas [BW94] (resp. various classes of regular expressions [BG96; BH99; FIS00], clock zones [AD94]). □

We do not require the preorder $\sqsubseteq$ to be a partial order (i.e., to be antisymmetric). Indeed, in predicate abstraction, we will design region structures with many distinct equivalent regions. Similarly, we do not require that the functions $\sqcup$ and $\sqcap$ be associative or commutative. This more general setting allows us to accommodate abstraction within the framework of region structures. However, for any finite set R' of regions, the operations $\sqcup R'$

and $\sqcap R'$ under any order of evaluation produce regions that are equivalent with respect to $\equiv$, and so there is no ambiguity.

2.1.2 Symbolic Abstraction Structures

A (*symbolic*) *abstraction structure* $\mathcal{A} = (\mathcal{R}, \widehat{\mathsf{post}}, \trianglelefteq)$ for a labeled transition system $\mathcal{S}$ consists of a computable region structure $\mathcal{R} = (R, \bot, \sqcup, \sqcap, \mathsf{post}, [\![\cdot]\!])$ for $\mathcal{S}$ along with a *precision* preorder $\trianglelefteq \subseteq R \times R$, and a computable total function $\widehat{\mathsf{post}} : R \to \Sigma \to R$ such that for all regions $r \in R$ and every label $l \in \Sigma$, we have:

$$\mathsf{post}.r.l \sqsubseteq \widehat{\mathsf{post}}.r.l$$

and $\widehat{\mathsf{post}}$ is monotonic with respect to the preorder $(\trianglelefteq \cap \sqsubseteq)$; that is, if $r \trianglelefteq r'$ and $r \sqsubseteq r'$, then

$$\widehat{\mathsf{post}}.r.l \trianglelefteq \widehat{\mathsf{post}}.r'.l$$
$$\widehat{\mathsf{post}}.r.l \sqsubseteq \widehat{\mathsf{post}}.r'.l$$

Hence, $\widehat{\mathsf{post}}$ is an overapproximation of the exact successor operator post on regions.

The novelty of this definition lies in the fact that regions carry precision information, which indicates how close the overapproximate operator $\widehat{\mathsf{post}}$ is to the exact operator post. In particular, for two $\equiv$-equivalent regions r and r', if $r \trianglelefteq r'$, then

$$\mathsf{post}.r.l \equiv \mathsf{post}.r'.l \sqsubseteq \widehat{\mathsf{post}}.r.l \sqsubseteq \widehat{\mathsf{post}}.r'.l$$

that is, $\widehat{\mathsf{post}}.r.l$ is a more precise overapproximation of the successor set than $\widehat{\mathsf{post}}.r'.l$. The precision preorder permits us to perform both the concrete operations of a labeled transition system $\mathcal{S}$ and abstract interpretation of $\mathcal{S}$ within a single region structure. A region r is no longer interpreted as simply a description of the *concrete* state set $[\![r]\!]$ of $\mathcal{S}$, but as a description of an *abstract* state set, for some abstract state space, whose concretization is $[\![r]\!]$. If $r \trianglelefteq r'$, then the abstract state space in which r is interpreted is more precise than the abstract state space of r'. The function $\widehat{\mathsf{post}}$ is extended to traces as: $\widehat{\mathsf{post}}.r.\epsilon = r$ and $\widehat{\mathsf{post}}.r.l\sigma) = \widehat{\mathsf{post}}.(\widehat{\mathsf{post}}.r.l).\sigma$.

2.1.3 Predicate Abstraction

Consider a labeled transition system $\mathcal{S} = (X, \Sigma, \rightsquigarrow, S_0)$. A *predicate language* $\mathcal{L}$ for $\mathcal{S}$ is a set of *predicates* that are interpreted over the states in $\mathsf{V}.X$ (*i.e.,* each predicate $p \in \mathcal{L}$ denotes a set $[\![p]\!] \subseteq \mathsf{V}.X$ of states), such that the following two conditions are satisfied.

1. The boolean closure of $\mathcal{L}$ is a decidable theory (*i.e.,* satisfiability is decidable).

2. The boolean closure of $\mathcal{L}$ is effectively closed under the (exact) successor operation in $\mathcal{S}$; that is, for every formula φ in the boolean closure of $\mathcal{L}$ and every label $l \in \Sigma$, we can compute the boolean combination $\varphi_l^{\mathsf{post}}$ of predicates from $\mathcal{L}$ such that: $[\![\varphi_l^{\mathsf{post}}]\!] = \{s' \mid \exists\ s \in [\![\varphi]\!].\ s \overset{l}{\rightsquigarrow} s'\}$

For a formula φ in the boolean closure of $\mathcal{L}$, and a set of predicates Λ, we define the *predicate abstraction*[GS97a] of φ w.r.t. Λ, written $\mathsf{Abs}.\Lambda.\varphi$ to be the *strongest* formula ψ (w.r.t. the implication order), in the boolean closure of Λ such that $\varphi \Rightarrow \psi$. Algorithms to compute $\mathsf{Abs}.\Lambda.\varphi$ are extensively studied in [GS97a] and then later in [DDP99; BPR01; FQ02; RSY04].

We define the abstraction structure $\mathcal{A}_\mathcal{L}$ for an $\mathcal{S}$ and $\mathcal{L}$ as follows:

A. Symbolic Region Structure. Let $\mathcal{R}_\mathcal{L} = (R, \bot, \sqcup, \sqcap, \mathsf{post}, [\![\cdot]\!])$. The regions in R are the pairs (φ, Π), where $\Pi : \Sigma \rightarrow 2^\mathcal{L}$ is a map from labels in Σ to finite sets of predicates called *support predicates*, and φ is a boolean formula over the predicates in the range of Π. The region $(\mathit{false}, \lambda l.\emptyset)$, represents $\bot$. The operators $\sqcup$ and $\sqcap$ are defined:

$$(\varphi, \Pi) \sqcup (\varphi', \Pi') = (\varphi \vee \varphi', \Pi \cup \Pi')$$

$$(\varphi, \Pi) \sqcap (\varphi', \Pi') = (\varphi \wedge \varphi', \Pi \cup \Pi')$$

where $\Pi \cup \Pi'$ is an abbreviation for $\lambda l.(\Pi.l \cup \Pi'.l)$.

Let $\mathsf{post}.(\varphi, \Pi).l = (\varphi_l^{\mathsf{post}}, \Pi')$, where Π' is the least superset of Π that contains all predicates in $\varphi_l^{\mathsf{post}}$. Finally, $[\![(\varphi, \Pi)]\!] = [\![\varphi]\!]$ straightforwardly interprets the boolean formula φ as a subset of $\mathsf{V}.X$; that is, $[\![(\varphi, \Pi)]\!]$ consists of all states in $\mathsf{V}.X$ that satisfy the constraint φ.

B. Parsimonious Predicate Abstraction Structure. We define the symbolic abstraction structure $\mathcal{A}_{\mathcal{L}}$ as $(\mathcal{R}_{\mathcal{L}}, \widehat{\mathsf{post}}, \trianglelefteq)$ where $\widehat{\mathsf{post}}$ and $\trianglelefteq$ are defined as follows. For a region (φ, Π) and a label $l \in \Sigma$, we define:

$$\widehat{\mathsf{post}}.(\varphi, \Pi).l = (\mathsf{Abs}.(\Pi.l).\varphi_l^{\mathsf{post}}, \Pi)$$

It is easy to check that $\widehat{\mathsf{post}}.(\varphi, \Pi).l$ is an overapproximation of $\mathsf{post}.(\varphi, \Pi).l$. For two regions (φ, Π) and (φ', Π'), we say $(\varphi, \Pi) \trianglelefteq (\varphi', \Pi')$ iff for each $l \in \Sigma$, $\Pi.l \supseteq \Pi'.l$.

Since $\mathcal{L}$ is a predicate language for $\mathcal{S}$, the preorder $\sqsubseteq$ on regions is computable. Moreover, from the definition of $\widehat{\mathsf{post}}$ it is clear that it can be computed effectively(as Abs can be computed effectively), that it is an overapproximation of post, and that it is monotonic with respect to the preorder $(\trianglelefteq \cap \sqsubseteq)$.

Intuitively, support predicates determine the current abstract state space, and the formula over the support predicates represents an abstract set of states. The support predicates indicate which predicates are important, *i.e.,* which predicates can be tracked by the abstract operation $\widehat{\mathsf{post}}$. Unlike traditional predicate abstraction, in parsimonious predicate abstraction we use different sets of predicates depending on the operation.

2.2 Imperative Programs

We first consider the analysis of imperative programs, written, *e.g.* in languages such as C. Our syntactic representation of such programs are *Control Flow Automata*(CFA), which are essentially control flow graphs[ASU86], with instructions labelling the edges instead of the vertices. In this work, we focus on C programs, but note that our methods can work on programs written in any language, so long as one can transform such programs into CFAs.

2.2.1 From Imperative Programs to LTSs

We first consider non-recursive imperative programs: we describe the syntax of such programs using CFAs, and then describe their semantics using LTSs.

Syntax

We consider a language with integer variables and pointers. Lvalues (memory locations) in the language are declared variables or dereferences of pointer typed expressions. The set

Types	$\tau \in \mathsf{Types}$	$::=$	$Int \mid \texttt{ref}\ \tau$
Lvalues	$\texttt{l} \in \mathsf{Lvals}.X$	$::=$	$\texttt{x} \mid *\texttt{l}$
			$x \in X$
Expressions	$\texttt{e} \in \mathsf{Exp}.X$	$::=$	$c \mid \texttt{l} \mid \texttt{e}_1 \oplus \texttt{e}_2$
			c is an integer constant
			$\oplus$ is a binary operator
Predicates	$\texttt{p} \in \mathsf{Pred}.X$	$::=$	$\texttt{e}_1 = \texttt{e}_2 \mid \texttt{e}_1 \leq \texttt{e}_2 \mid \neg\texttt{p} \mid \texttt{p}_1 \wedge \texttt{p}_2 \mid \texttt{p}_1 \vee \texttt{p}_2$
Operations	$\texttt{op} \in \mathsf{Op}.X$	$::=$	$\texttt{l:=e} \mid \texttt{assume}[\texttt{p}] \mid \texttt{f}() \mid \texttt{return}$

Figure 2.1: Program Syntax

Types is a set of non-recursive types for variables. For a set X of statically typed variables:

$\mathsf{Lvals}.X$ is the set of Lvalues corresponding to memory locations accessible from X.

$\mathsf{Exp}.X$ is the set of arithmetic expressions over the variables X,

$\mathsf{Pred}.X$ is the set of boolean expressions (boolean closure of arithmetic and pointer comparisons) over X, and,

$\mathsf{Op}.X$ is the set of operations over X comprising:

- *Assignments* $\texttt{l := e}$, where $\texttt{l} \in \mathsf{Lvals}.X$ and $\texttt{e} \in \mathsf{Exp}.X$,
- *Assumes predicates* $\texttt{assume}[\texttt{p}]$, where $\texttt{p} \in \mathsf{Pred}.X$, representing a condition that must be true for the edge to be taken,
- *Function calls* $\texttt{f}()$, where f is a function, and,
- *Return* statements $\texttt{return}$.

The above are summarized in Figure 2.1. In the non-recursive setting, there is no need for functions to have local variables, so we omit them from the exposition for clarity.

A *control flow automaton* $C = (X, PC, pc_0, \rightarrow)$ comprises: (1) a set of variables X, (2) a set of control locations PC, with an initial control location $pc_0 \in PC$, (3) a finite set of directed edges labeled with operations $\rightarrow \subseteq PC \times \mathsf{Op}.X \times PC$. An edge $(pc, \mathsf{op}, pc') \in \rightarrow$ is written $pc \xrightarrow{\mathsf{op}} pc'$.

A *Program* $P = (F, f_{\texttt{main}}, \mathsf{typ})$ comprises (1) a set of functions F, where each function $f \in F$ is denoted by its CFA C_f, (2) a special initial function $f_{\texttt{main}}$, and, (3) a map typ from $P.X$ to Types where $P.X$ are the (data) variables of P, *i.e.*, the union the variables of

```
 Example() {
1:  if (*){
7:     do {
            got_lock = 0;
8:          if (*){
9:             lock();
               got_lock++;
            }
10:         if (got_lock){
11:            unlock();
            }
12:    } while (*)
    }
2:  do {
         lock();
         old = new;
3:       if (*){
4:          unlock();
            new++;
         }
5:  } while (new != old);
6:  unlock();
    return;
}
```

```
lock(){
13:if (LOCK == 0){
14:   LOCK = 1;
   } else {
15:   err()
   }
}

unlock(){
16:if (LOCK == 1){
17:   LOCK = 0;
   } else {
18:   err()
   }
}

err(){
ERR:
}
```

Figure 2.2: C program

the CFAs of the functions of prog, *i.e.*, $P.X = \bigcup_{f \in F} C_f.X$. The *locations* of a program P written $P.PC$ are the set $\bigcup_{f \in P.F} C_f.PC$. The *commands* of a program P written $P.Cmd$ are the pairs (op, pc) such that P has a function f such that C_f has an edge $\cdot \xrightarrow{\mathsf{op}} pc$.

We extend the map typ to lvalues by letting $\mathsf{typ}.*\mathtt{l}_1$ equal τ if $\mathsf{typ}.\mathtt{l}_1$ is $\mathtt{ref}\ \tau$, otherwise $\mathtt{l}$ is not well typed w.r.t. P. We extend the map typ to expressions by letting $\mathsf{typ.e}$ equal (1) *Int* if e is either an integer constant c or of the form $\mathsf{e}_1 \oplus \mathsf{e}_2$ and $\mathsf{typ.e}_1$, and $\mathsf{typ.e}_2$ are *Int*, (2) $\mathsf{typ}.\mathtt{l}$ if e is the lvalue $\mathtt{l}$, and, (3) otherwise e is not well typed w.r.t. P. A formula p in $\mathsf{Pred}.X$ is well typed w.r.t. P if for each occurrence of $\mathsf{e}_1 = \mathsf{e}_2$ inside the formula, we have $\mathsf{typ.e}_1 = \mathsf{typ.e}_2$, for each occurrence of $\mathsf{e}_1 \leq \mathsf{e}_2$ we have $\mathsf{typ.e}_1 = \mathsf{typ.e}_2 = Int$.

Example 2 [Programs and CFAs] Consider the program given in Figure 2.2. The initial function is `example()`. The functions `lock()` and `unlock()` control a variable called `LOCK`, which is 1 when the lock is held by the function `Example`, and 0 otherwise. The CFA for the function `Example` is shown in Figure 2.3. The labels in the C program correspond to the automaton vertices with the same label. The edges labeled with boxes are basic blocks. With slight abuse of notation, we combine the operations `lock()` and the assignment `old`

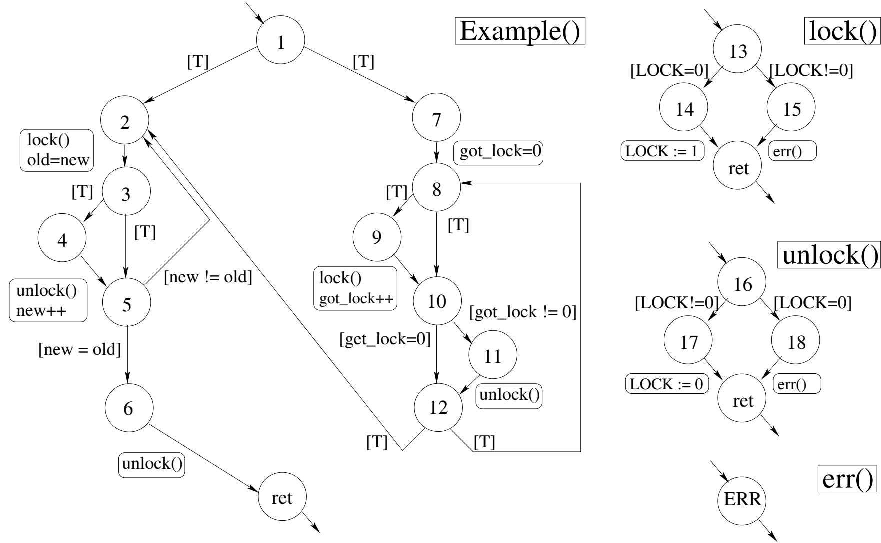

Figure 2.3: Control flow automaton

`:= new` in a single command (edge 2→3), indicating two successive edges: the first from 2 to a vertex 2′ (not shown for clarity), corresponding to the call, and the second from 2′ to 3, corresponding to the assignment. The edges labeled with [·] correspond to assume predicates. The `if (*)` represents branches that are taken due to predicates that are not modeled; we assume that either direction can be taken, hence both outgoing edges are labeled with `[T]`, which stands for the assume predicate *true*. □

Semantics

We define an LTS corresponding to a program P, written $\mathcal{S}_P$ as follows. For clarity, we shall first consider the class (PI) of programs that are pointer-free, *i.e.,* where all variables are of type *Int*, and then generalize to programs with pointers (PII).

Variables. The set of variables of $\mathcal{S}_P$ are: (1) a “program counter” whose value is an element of $P.PC$, (2) a “store” variable whose value is a map from $P.X$ to integers, and, (3) a “call stack” whose value is an element of $P.PC^*$. We write a state of $\mathcal{S}_P$ as a triple (pc, m, cs), where: (1) pc is the value of the program counter, (2) m is a map

from $P.X$ to integers, *i.e.,* a valuation for the data variables of P, and, (3) cs is the value of the call stack. The (integer) value of a program variable $x \in P.X$ in the store m is written $m.x$. This is extended naturally to (integer) expressions $e \in \mathsf{Exp}.(P.X)$, and (boolean) predicates $p \in \mathsf{Pred}.(P.X)$.

Labels. The set of labels of $\mathcal{S}_P$ are the set of all *commands* of P.

Transitions. We say that $(pc, m, cs) \overset{l}{\rightsquigarrow} (pc', m', cs')$ where:

$$(pc', m', cs') = \begin{cases} (C_f.pc_0, m, pc_l \cdot cs) & \text{if } l \text{ is a function call } (\texttt{f()}:pc_l), \\ (pc'', m, cs'') & \text{if } l \text{ is } (\texttt{return}:\cdot) \text{ and } cs = pc'' \cdot cs'', \\ (pc_l, m, cs) & \text{if } l \text{ is } (\texttt{assume[p]}:pc_l) \text{ and } m.p \text{ is true}, \\ (pc_l, m[x \mapsto m.e], cs) & \text{if } l \text{ is } (\texttt{x:=e}:pc_l). \end{cases}$$

Initial State. The initial states are $S_0 = \{s_0\}$ where

$$s_0 = ((P.f_{\texttt{main}}).pc_0, m_0, \epsilon)$$

, where m_0 is a map that sets all integer variables to 0.

LTSs for PII. To generalize the above to programs with pointer variables, we generalize the definition of stores as follows. Define *Addr* to be an infinite set of addresses, that contains as a subset all the variables, *i.e.,* for every variable x there is an address in *Addr*, that by abuse of notation, we call x. A store is generalized to be a map from *Addr* to *Addr* $\cup$ *Int*. The value of an lvalue l in a store m, written $m.l$, is $m.(m.l')$ if l is the dereference $*l'$, and $m.x$ if l is the variable x. As before we can naturally extend stores to expressions in $\mathsf{Exp}.(P.X)$ and predicates in $\mathsf{Pred}.(P.X)$. The definition of $\mathcal{S}_P$ is the same as for PI programs with the new definition of stores, only for assignments $*\texttt{l} := \texttt{e}$, the store m' equals $m[m.l \mapsto m.e]$.

2.2.2 Predicate Abstraction for Imperative Programs

We use the predicate-abstraction framework described in Section 2.1.3. As before, we first discuss PI programs, and then generalize to PII programs.

Predicate Language

Let *FOL* be the set of formulas in the first-order logic of linear equality and uninterpreted functions, shown in Figure 2.4. An atomic predicate is an inequality of the form $0 \leq x$, where

Terms	$t \in TERMS$	$::=$	$cx \mid t_1 + t_2 \mid f^k(t_1, \ldots, t_k)$ c is an integer constant, x is a variable, f^k is a $k-$ary uninterpreted function
Atomic Predicate	$p \in ATOMS$	$::=$	$0 \leq t$
Predicate Language	$\varphi \in FOL$	$::=$	$p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2$

Figure 2.4: Predicate Language *FOL*

x is a term. We can model propositional variables using integers and comparison. A *literal* is either an atomic predicate or its negation. A *clause* is a disjunction of literals. A *cube* is a conjunction of literals. For three terms t, t_1, t_2 and a boolean formula φ in FOL, we write $t = \mathsf{ite}.\varphi.t_1.t_2$ as an abbreviation for $(\varphi \wedge t = t_1) \vee (\neg\varphi \wedge t = t_2)$. For three formulas $\varphi, \varphi_1, \varphi_2$ in FOL we write $\mathsf{ite}.\varphi.\varphi_1.\varphi_2$ as an abbreviation for $(\varphi \wedge \varphi_1) \vee (\neg\varphi \wedge \varphi_2)$. For a store m that is a map from X to Int, and predicate $\varphi \in FOL$, we say $m \models \varphi$ if $\varphi \wedge \left(\bigwedge_{x \in X} x = m.x\right)$ is satisfiable in FOL. Our predicate language $\mathcal{L}$ is FOL. Notice that FOL (even when enriched with the theory of arrays) is decidable, and efficient decision procedures are available [Nel81; BHJ+].

Symbolic Abstraction Structure

We now define the Symbolic Abstraction Structure corresponding to predicate abstraction for imperative programs. First, we define the Symbolic Region Structure, and then the corresponding Abstraction Structure.

A. Symbolic Region Structure.

Data Regions. A *data region* is a pair (φ, Π), where Π is a map from the commands of P, to finite subsets of $\mathcal{L}$ and φ is a boolean formula over predicates in the range of Π. We write D for the set of all data regions. Data regions model the data components (*i.e.,* the valuations of the variables) of states; the other components *i.e.,* the program counter and the call stack are modeled explicitly.

Atomic Regions. An *atomic region* is a triple consisting of a control location, a data region, and a call stack. We write $A = PC \times D \times PC^*$ for the set of atomic regions. For every atomic region $(pc, (\varphi, \Pi), cs) \in A$, we define

$$[\![(pc, (\varphi, \Pi), cs)]\!] = \{(pc, m, cs) \mid m \models \varphi\}$$

For two atomic regions $a = (pc, (\varphi, \Pi), cs)$ and $a' = (pc', (\varphi', \Pi'), cs')$, we define:

1. $a \sqcup a'$ as $(pc, (\varphi \vee \varphi', \Pi \cup \Pi'), cs)$ if $pc = pc'$ and $cs = cs'$ and $\bot$ otherwise,

2. $a \sqcap a'$ as $(pc, (\varphi \wedge \varphi', \Pi \cup \Pi'), cs)$ if $pc = pc'$ and $cs = cs'$ and $\bot$ otherwise, and,

3. $a \sqsubseteq a'$ if $pc = pc'$ and $cs = cs'$ and $\varphi \Rightarrow \varphi'$ (*i.e.,* $\varphi \wedge \neg\varphi'$ is not satisfiable).

Regions. A *region* r is a *finite* set of atomic regions. We write $R \subset 2^A$ for the set of regions. For every region r, we define $[\![r]\!] = \cup_{a \in r}[\![a]\!]$. For two regions r, r', we define:

1. $r \sqcup r'$ as the set $r \cup r' \cup \{a \sqcup a' \mid (a, a') \in r \times r'\}$,

2. $r \sqcap r'$ as the set $\{a \sqcap a' \mid (a, a') \in r \times r'\}$, and,

3. $r \sqsubseteq r'$ if for every $a \in r$ there exist some $a'_1, \ldots, a'_k \in r'$ such that $a \sqsubseteq a'_1 \sqcup \ldots \sqcup a'_k$.

Strongest Postconditions. For a formula φ and a command l, the formula $\mathsf{SP}.\varphi.l$ denotes the *strongest postcondition* [Dij76; Gri81] of φ w.r.t l; that is, $\mathsf{SP}.\varphi.l$ is the strongest formula whose truth holds after l terminates, given that φ was true before l was executed.

$$\mathsf{SP}.\varphi.l = \begin{cases} \varphi & \text{if } l \text{ is a function call or a return,} \\ \varphi \wedge p & \text{if } l \text{ is } (\texttt{assume}[\texttt{p}]{:}\cdot), \\ \exists x'.\varphi[x'/x] \wedge x = e[x'/x] & \text{if } l \text{ is the assignment } (\texttt{x:=e}{:}\cdot). \end{cases}$$

We generalize the strongest postcondition to data regions, by defining the operator $\mathsf{SP}^{\mathsf{D}} : D \to \Sigma \to D$ as:

$$\mathsf{SP}^{\mathsf{D}}.(\varphi, \Pi).l = (\mathsf{SP}.\varphi.l, \Pi')$$

where $\Pi'.l$ is the least superset of $\Pi.l$ that contains all predicates in $\mathsf{SP}.\varphi.l$.

Post. We define the operator $\mathsf{post} : A \to \Sigma \to A$ as $\mathsf{post}.(pc, d, cs).l = (pc', d', cs')$ where $d' = \mathsf{SP}^{\mathsf{D}}.d.l$ and:

$$(pc', cs') = \begin{cases} (C_f.pc_0, pc_l \cdot cs) & \text{if } l \text{ is a function call } (\texttt{f}(){:}pc_l), \\ (pc'', cs'') & \text{if } l \text{ is } (\texttt{return}{:}\cdot) \text{ and } cs = pc'' \cdot cs'', \\ (pc_l, cs) & \text{otherwise } l \text{ is } (\cdot{:}pc_l). \end{cases}$$

We generalize post to regions by defining: $\mathsf{post}.r.l = \{\mathsf{post}.a.l \mid a \in r\}$.

Example 3 [Regions,Post] Consider the program from Example 2. Let Π be any set of support predicates, let $\varphi = \mathit{LOCK} = 0 \wedge \mathit{new} = \mathit{old}$, let the region $r = (2, (\varphi, \Pi), \epsilon)$, and let

$l_1 = (\texttt{unlock}() : 2')$, $l_2 = (\texttt{assume}[LOCK! = \texttt{0}] : 17)$, $l_3 = (\texttt{LOCK} := \texttt{0} : ret)$, $l_4 = (\texttt{return} : \cdot)$, and, $l_5 = (\texttt{new} := \texttt{new} + \texttt{1} : 3)$.

We have $\mathsf{SP}.\varphi.l_2 = \varphi \wedge LOCK = 0$, which is just φ.

$$\text{Next, } \mathsf{SP}.\varphi.l_3 = \exists LOCK'.LOCK' \neq 0 \wedge new = old \wedge LOCK = 0$$

which we shall call φ'.

Finally, as l_1 is a call and l_4 its matching return,

$$\begin{aligned}
\mathsf{SP}.\varphi.(l_1; l_2; l_3; l_4; l_5) &= \mathsf{SP}.\varphi.(l_2; l_3; l_5) \\
&= \mathsf{SP}.(\mathsf{SP}.\varphi.(l_2; l_3)).l_5 = \mathsf{SP}.\varphi'.l_5 \\
&= \exists new', LOCK'. \\
&\qquad LOCK' \neq 0 \wedge new = old \wedge LOCK = 0 \wedge new = new' + 1
\end{aligned}$$

which we shall call φ''.

Hence, $\mathsf{post}.r.(l_1 l_2 l_3 l_4) = (3, (\varphi'', \Pi'), \epsilon)$ where Π' has the updated support predicates.

□

B. Predicate Abstraction Structure. We now define the precision preorder $\trianglelefteq$ and the abstract post operator $\widehat{\mathsf{post}}$.

Precision Preorder. For a region $r \in R$ we define $r.\Pi = \bigcup_{(\cdot,(\cdot,\Pi),\cdot) \in r} \Pi$. We say that $r_1 \trianglelefteq r_2$ if for each command l of P, $r_1.\Pi.l \supseteq r_2.\Pi.l$.

Abstract Postcondition. For a predicate map Π a formula φ and a command l, $\mathsf{SP}_\Pi.\varphi.l$ is the Π-abstract postcondition of φ w.r.t. l, defined as:

$$\mathsf{SP}_\Pi.\varphi.l = \mathsf{Abs}.(\Pi.l).(\mathsf{SP}.\varphi.l)$$

We generalize this to data regions by defining the operator $\widehat{\mathsf{SP}^\mathsf{D}} : D \to \Sigma \to D$ as:

$$\widehat{\mathsf{SP}^\mathsf{D}}.(\varphi, \Pi).l = (\mathsf{SP}_\Pi.\varphi.l, \Pi)$$

Note that we *do not* add all the predicates in $\mathsf{SP}_\Pi.\varphi.l$. We define $\widehat{\mathsf{post}}$ exactly as we define post, using $\widehat{\mathsf{SP}^\mathsf{D}}$ in the place of SP^D. We define the operator $\widehat{\mathsf{post}}$ operator exactly the same way as we defined post, but using an abstract postcondition operator instead of the strongest postcondition.

Example 4 [Abstract Post] Consider the program, φ, σ, r from Example 3. Let the set of support predicates Π of r be $LOCK = 0$, $LOCK = 1$, and $new = old$. We have:

$$\begin{aligned}
\text{We have } \mathsf{SP}_{\Pi}.\varphi.l_2 &= \mathsf{Abs}.(\Pi.l_2).(\mathsf{SP}.\varphi.l_2) = \mathsf{Abs}.(\Pi.l_2).\varphi = \varphi \\
\text{Next, notice } \mathsf{SP}_{\Pi}.\varphi.l_3 &= \mathsf{Abs}.(\Pi.l_3).(\mathsf{SP}.\varphi.l_3) \\
&= \mathsf{Abs}.(\Pi.l_3).(\exists LOCK'.lockb' \neq 0 \wedge new = old \wedge LOCK = 0) \\
&= new = old \wedge LOCK = 0, \text{ which we call } \varphi'. \\
\text{Again, } \mathsf{SP}_{\Pi}.\varphi'.l_5 &= \mathsf{Abs}.(\Pi.l_5).(\mathsf{SP}.\varphi'.l_5) \\
&= \mathsf{Abs}.(\Pi.l_5).(\exists new'.new' = old \wedge LOCK = 0 \wedge new = new' + 1) \\
&= \neg(new = old) \wedge (LOCK = 0), \text{ which we call } \varphi''.
\end{aligned}$$

As before $\mathsf{SP}_{\Pi}.\varphi.(l_1; l_2; l_3; l_4; l_5) = \mathsf{SP}_{\Pi}.\varphi.(l_2; l_3; l_5) = \varphi''$

$$\text{Hence, } \widehat{\mathsf{post}}.r. = (3, (\varphi'', \Pi), \epsilon).$$

□

Programs with Pointers

To generalize the above to PII programs it suffices to generalize the predicate language *FOL* to model stores and to generalize the strongest postcondition.

Stores. The classical way to model the store is to use memory expressions and the theory of arrays [MP67; Nel81; Nec97b; FLL+02], which comes equipped with *memory* variables and two special functions, *sel* and *upd*. The function *sel* takes a memory M and an address a and returns the contents of the address a; the function *upd* takes a memory M, an address a, and a value v, and returns a new memory that agrees with M except that the address a now has value v. The relationship between *sel* and *upd* is succinctly stated by McCarthy's axiom [MP67]:

$$sel(upd(M, a, v), b) = \mathsf{ite}.(a = b).v.sel(M, b)$$

In the sequel, we assume that *FOL* contains this axiom in addition to the axioms for equality, uninterpreted functions and linear arithmetic.

The formulas of *FOL* that we use to denote regions, contain the special memory variable M. For a store m, *i.e.,* a map from *Addr* to $Addr \cup Int$, and predicate $\varphi \in FOL$, we say

$m \models \varphi$ if $\varphi \wedge \left(\bigwedge_{l \in Addr} sel(M,l) = m.l\right)$ is satisfiable in *FOL*, where each l is a variable denoting an address in *Addr*.

Strongest Postconditions. For a memory variable M and a variable $x \in X$, define $M.x = sel(M,x)$, and for an lvalue $*l$, define $M.(*l) = sel(M, M.l)$. With some slight abuse of notation, we use M in this way to denote a map from lvalues to memory expressions over M. We can extend this map to expressions in $\mathsf{Exp}.X$ and predicates in $\mathsf{Pred}.X$. In our predicate language every lvalue l is an abbreviation for the memory expression $M.l$. The new definition of the strongest postcondition is:

$$\mathsf{SP}.\varphi.l = \begin{cases} \varphi & \text{if } l \text{ is a call or a return,} \\ \varphi \wedge M.p & \text{if } l \text{ is } (\texttt{assume}[\texttt{p}] : \cdot), \\ \exists M'.\varphi[M'/M] \wedge M = upd(M', M'.l, M'.e) & \text{if } l \text{ is an assignment } (\texttt{l:=e} : \cdot). \end{cases}$$

It is easy to check that the Region Structure and Predicate Abstraction Structure satisfy the requirements of the previous Section 2.1, with the following caveat: our choice of predicate language means that we can analyze exactly only C programs whose basic data types are integers and pointers, with the operations integer addition, arithmetic comparison, and pointer equality. In the sequel, we assume that the given C program has been modified so that any C data type or operation that we cannot model in our predicate language (*e.g.* multiplication) has been replaced by an uninterpreted function. This modification is conservative: a behavior of the original program is still a behavior in the modified program (but there may be more behaviors in the modified program, *e.g.* behaviors arising due to branches that are satisfiable as the multiplication operator is uninterpreted). For the code we have analyzed (see Chapter 4.1), the properties of interest can all be proved in this way. For other programs or properties, of course, one may have to use richer predicate languages.

2.3 The Safety Verification Problem

We define the *reachable* states of an LTS $\mathcal{S}$ as: For a subset S of the X-states, we define the reachable set as:

$$\mathsf{Reach}.\mathcal{S}.S = \{s' \mid s \overset{\sigma}{\rightsquigarrow} s' \text{ for some } s \in S, \sigma \in \Sigma^*\}$$

Given an LTS $\mathcal{S}$, with initial states S_0, and a set of error states $\mathcal{E}$, we say $\mathcal{S}$ is safe w.r.t. $\mathcal{E}$ iff $\mathsf{Reach}.\mathcal{S}.S_0 \cap \mathcal{E} = \emptyset$. The *Safety Verification Problem* is to decide if $\mathcal{S}$ is safe w.r.t. $\mathcal{E}$.

Counterexamples

For a region r_0, a trace σ is: (1) *feasible* from r_0 if $\mathsf{post}.r_0.\sigma \not\equiv \bot$, and, (2) *abstractly feasible* from S_0 if $\widehat{\mathsf{post}}.r_0.\sigma \not\equiv \bot$. For a region $\mathcal{E}$, the trace σ is a: (1) *counterexample* to $\mathcal{E}$ from r_0 if $\mathsf{post}.r_0.\sigma \sqcap \mathcal{E} \not\equiv \bot$, and, (2) *abstract counterexample* to $\mathcal{E}$ from r_0 if $\widehat{\mathsf{post}}.r_0.\sigma \sqcap \mathcal{E} \not\equiv \bot$.

If $\mathsf{post}.r_0.\sigma \sqcap \mathcal{E}$ is not $\bot$ *i.e.,* not empty, then there exist $s_0 \in [\![r_0]\!]$ and $s_{\mathcal{E}} \in \mathcal{E}$ such that $s_0 \overset{\sigma}{\rightsquigarrow} s_{\mathcal{E}}$, and hence:

Proposition 1 *For an LTS $\mathcal{S}$ and a region structure $\mathcal{R}$ for it, if a trace σ is a counterexample to $\mathcal{E}$ from r_0, where $[\![r_0]\!] = \mathcal{S}.S_0$, then $\mathcal{S}$ is not safe w.r.t $\mathcal{E}$.*

A trace σ that is an abstract counterexample to $\mathcal{E}$ from r_0 is called *genuine* if it is also a counterexample to $\mathcal{E}$ from r_0, otherwise it is called *bogus.*

Imperative Programs

When verifying an imperative program, the set of initial states $[\![r_0]\!]$ is given by the region, $r_0 = (pc_0, (\varphi, \emptyset), \epsilon)$ where pc_0 is the start location of the function $f_{\texttt{main}}$ at which execution begins, φ is a precondition that holds before execution begins, and ϵ is the empty call stack. The set of error states $[\![\mathcal{E}]\!]$ is given by a special error location in the program $pc_{\mathcal{E}}$. Hence the error states are, given by the region, $\mathcal{E} = (pc_{\mathcal{E}}, (\mathit{true}, \cdot), \cdot)$, where $\cdot$ denotes that those values can be anything.

Example 5 [Safety Verification] Consider the C program from Figure 2.2. The property we wish to check is that (1) the function `Example` never calls `lock` when it holds the lock, and (2) it never calls `unlock` when it does not hold the lock. We assume the precondition that when the function `Example` starts, it does not hold the lock, *i.e., LOCK* is 0. We have instrumented the functions `lock()` and `unlock()` so that checking this property amounts to checking that the *ERR* label is never reached in the code, when the precondition $LOCK = 0$ holds before the program begins. Hence, the safety verification problem is to check whether `Example` is safe w.r.t. $(pc_{\mathcal{E}}, (\mathit{true}, \cdot), \cdot)$ (from the initial states $r_0 = (pc_1, (LOCK = 0, \emptyset), \epsilon)$).
□

Traces correspond to paths through the CFAs. Let the region $r_0 = (pc_0, (\varphi, \Pi), \epsilon)$. A trace σ is feasible (resp. abstractly feasible) from r_0 if $\mathsf{SP}.\varphi.\sigma$ (resp. $\mathsf{SP}_{\Pi}.\varphi.\sigma$) is satisfiable.

```
(assume[T] : 2);
(lock(); old := new : 3);
(assume[T] : 4);
(unlock(); new := new + 1 : 5);
(assume[new = old] : 6);
(unlock() : ·);
(assume[LOCK = 0] : 18);
(err() : ·)
```

Figure 2.5: Trace

For imperative programs, the set $\mathcal{E}$ is $(pc_{\mathcal{E}}, (true, \cdot), \cdot)$. Hence counterexamples correspond to traces starting at pc_0 and ending at $pc_{\mathcal{E}}$. Such traces are counterexamples (resp. abstract counterexamples) to $\mathcal{E}$ from r, if they are feasible (resp. abstractly feasible), from r_0, and they are bogus counterexamples if they are abstractly feasible but not feasible from r_0.

Example 6 [Bogus Counterexample] Consider the trace in Figure 2.5: This trace is infeasible, and hence not a counterexample, as after the second and fourth commands, the value of `new` is different from the value of `old`, and so the fifth command cannot execute. This trace is abstractly feasible from the region r_0, as $\mathsf{SP}_{\emptyset}.r.l$ is *true* for all r, l. Hence, this trace is a bogus counterexample. □

Chapter 3

Lazy Abstraction

Given a program, an *initial* state from which the program begins execution, a set of *error states*, the safety verification problem is to decide whether there is an execution of the program that leads it from the initial state to an error state.

When the program is given as a finite directed graph, where the initial and error states are vertices, and steps of the program's execution edges, the problem can be solved simply by traversing the graph to find whether an error vertex is reachable from the initial vertex. Unfortunately, the difficulty of writing interesting programs in this manner greatly outweighs the ease by which they may be analyzed. Thus, most programs are written in languages like C or Java, and, as they are written using variables, pointers, procedures and threads, they do not correspond to graphs small enough to be traversed.

The *Counterexample-Guided Abstraction Refinement* (CEGAR) approach to the safety verification problem is to iterate the following three steps [Kur94; CGJ+00; BR02b].

First, an abstract, finite-state model, an abstraction, of the program is constructed using an appropriate abstract domain. For hardware circuits one may construct an abstraction by picking a small subset of the variables considered relevant to the property being checked, and treating the other variables as unknown or "don't care" [Kur94; CGJ+00]. For C programs one may construct an abstraction by picking a finite set of *predicates* that capture relationships between program variables, and then constructing a *boolean program*(BP) made up of boolean variables corresponding to the predicates. For each statement of the program, the BP boolean program updates its variables by finding how executing the statement would affect the truth of the

predicates [BMMR01].

Second, we *model check* the finite-state abstraction, by exhaustively exploring its state space, to see if an error state is reached. As the abstraction overapproximates the behaviours of the real or concrete program, the fact that the abstraction never visits an error state implies that the program never visits one either, and so is safe. If, on the other hand, the abstraction hits an error, then we have an abstract counterexample, *i.e.,* a path in the abstraction from the abstract initial state to an abstract error state.

Third, we check if the abstract counterexample corresponds to a path in the real counterexample *i.e.,* a path to an error state from an initial state in the real program. If so, the program is reported to be unsafe. If not, the abstract domain used in the first step, *e.g.* the set of relevant variables, or relevant predicates is refined, *e.g.* by adding variables or predicates to the set, and we repeat the loop from the first step.

The main hurdle to making this scale to large programs is the state explosion problem: the number of abstract states is exponential in the number of predicates or boolean variables. Hence, both constructing and analyzing the abstract program are very computationally expensive.

Our approach, *Lazy Abstraction*, to circumvent the state-explosion problem, is to never construct an abstract program, by to tightly integrate the three steps of the loop in a way that:

1. The abstraction is built *on-the-fly*, so that only reachable states, which are often a tiny fraction of the entire, exponentially large, state space, are abstracted.

2. The refinement is *local*, so that only the small part of the abstraction through which the abstract counterexample path passes is refined, and re-analyzed, while other parts that have little to do with the counterexample and which are known to be safe using a coarse abstraction, are left untouched.

3. The abstraction is *parsimonious*, in that different parts of the state space use different abstractions, namely they are only as precise as is required to verify that part of the system. Homogenous abstractions suffer as the number of states is exponential in the

total number of predicates used, instead of just in the number of predicates used per location of the program.

Lazy Abstraction is made up of two ingredients which give it the above properties.

1. Reachability Trees. The first ingredient is the use of *reachability trees* to explore the state space. A reachability tree has nodes labelled by formulas describing a set of program states, and edges labelled by a program operations. A child node is labelled by an overapproximation (with respect to a set of predicates) of the states the program can be in, if it was in one of the states corresponding to the parent's label, and executed the operation labelling the edge. Such a tree can be thought of as an unrolling of the Control-flow Graph of the program. The tree is finite as whenever we see a node whose label has been seen before, we stop unrolling from that node, and as a finite set of predicates is tracked, the number of distinct labels is finite. When we hit an error state, the path in the tree from the root to the error state is a counterexample. We find the pivot node, which corresponds to a node on the path from which the suffix to error has no concrete counterpart and we refine the subtree of the pivot node. Reachability trees enable us to abstract the reachable states, as the states corresponding to node labels are all abstractly reachable, and the allow local refinement, as we only refine the subtree of the pivot node.

2. Parsimonious Abstractions. The second ingredient is the use of *parsimonious abstractions*. The number of predicates that one needs to track grows with the size of the program being analyzed. However, most predicates are only locally useful, *i.e.,* only useful when analyzing certain parts of the program, and irrelevant in others. If locality is not exploited, then the sheer number of facts may render the abstract system too detailed to be amenable to analysis, as the size of the abstract system grows exponentially with the number of predicates. Consider the program fragment shown in Figure 3.1. The property we wish to check is that locking and unlocking alternate, *i.e.,* between any two calls of `lock()` there must be a call of `unlock()`, and between any two calls of `unlock()` there must be a call of `lock()`.

A static analysis that tracks whether or not the lock is held returns false positives, *i.e.,* error traces that arise from the imprecision of the analysis. One such spurious error

```
      while(*){
1:      if (p1) lock ();          assume[p1];
        if (p1) unlock ();        lock ();
        ...                       assume[¬p1];
2:      if (p2) lock ();          assume[p2];
        if (p2) unlock ();        lock ();
        ...
n:      if (pn) lock ();
        if (pn) unlock ();
      }
```

Figure 3.1: Program; spurious counterexample.

trace is shown on the right in Figure 3.1. The analysis is fooled because it does not track the predicate $p1$ which correlates the first two `if` statements; either both happen or neither happens, and either way the error cannot be reached. Similar counterexamples show that all of the predicates $p1, \ldots, pn$ must be tracked, but as a result, the analysis blows up, because it is not clear when we can "merge" states with different predicate values, and without merging there are an exponential number of states. [1] Notice however that in this program, each predicate is only locally useful, *i.e.,* each `pi` is "live" only at the statements between labels `i` and (not including) $\mathtt{i}+\mathtt{1}$. Hence, we need a method that infers both the predicates and *where* the predicates are useful. Our refinement algorithm infers the predicates `pi` and also that `pi` is useful only between the labels `i` and $\mathtt{i}+\mathtt{1}$; outside these labels, we can forget the value of `pi`. Thus our analysis considers, in this example, only a linear number of distinct states.

More generally, instead of tracking a monolithic set of predicates, we maintain a map from program locations to predicates, and at each location, we use only the predicates relevant at that location, and replace the values of the other predicates with "don't cares". In our experience, many large software systems have the property that, while the number of relevant predicates grows with the size of the system, each predicate is useful only in a small part of the state space, *i.e.,* the number of predicates that are relevant at any particular program location is small. By exploiting this property one can make a precise analysis scale to large programs.

[1] For this particular example certain state representation methods such as BDDs would implicitly merge the states.

In the rest of this chapter, we first give, in Section 3.1, an overview of Lazy Abstraction using the example from Figure 2.2; while Lazy Abstraction is a generic technique that works on any modeling paradigm, we provide examples from the automatic verification of C programs. Then we present the algorithm by presenting, in Section 3.2, the first ingredient, abstract state exploration via reachability trees, followed by, in Section 3.3, the second ingredient, our algorithm for refining abstractions, by enriching the set of predicates tracked at each location. Then in Section 3.4 we present two theoretical results related to Lazy Abstraction. The main question of interest is, of course, given a theory of predicates (such as Presburger arithmetic), a C program, and a correctness property, if there is a *finite* set of predicates that contains enough information for verifying the program (*i.e.,* if there is a predicate abstraction that is finite-state and witnesses the correctness property). We show this question to be, not surprisingly, undecidable. It follows that Lazy Abstraction (or any method) must be a *semi-algorithm*, which may or may not terminate. However, we show that the Lazy Abstraction semi-algorithm terminates under a customary condition on the predicate theory (no infinite ascending chains of predicates) and an abstract condition on the program (finite trace equivalence), which has been established for many interesting classes of infinite-state systems (such as timed automata [AD94]). The chapter concludes with a discussion of related work.

3.1 A Locking Example

We begin by showing how lazy abstraction works on C programs, using the program from Example 5. The algorithm works in two phases. The first is the forward-search phase, where we build the reachability tree, that represents the reachable, abstract state space of the program. Each edge of the tree is labeled by a program fragment, such as a basic block of assignments or an assume predicate. We call the path from the root of the tree to the node, the *node's path*, and we call the sequence of commands labelling the node's path the *node's trace*. Each node of the tree is labeled by: (1) a predicate map, called the node's map, that maps each command of the program to a (finite set of predicates), which determines the precision of the abstraction, (2) a program location, called the node's location, corresponding to the location the program is in after executing the node's trace,

(3) a boolean combination of the predicates, called the node's region, which describes the state of the program's variables after executing the node's trace.

If we find that an error state is reachable in the tree, then we go to the second phase, which checks if the error is real or results from our abstraction being too imprecise (*i.e.,* if we lost too much information by restricting ourselves to a particular set of predicates). If the latter is true, we ask a theorem prover for additional predicates, such that by using the new predicates we can rule out that particular

spurious counterexample (and maybe others as well). However, we add the new predicates only to those nodes in the search tree where they are required.

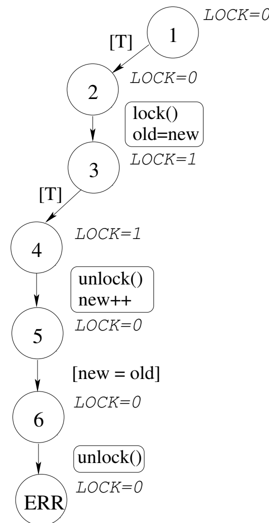

Figure 3.2: Forward Search

3.1.1 Verification

The model checking is done on the CFA shown in Figure 2.3. For simplicity, we assume that the call to `lock()` and `unlock()` are atomic commands: if `lock()` is called properly (*i.e.,* , with the lock not held), then it sets the value of $LOCK$ to 1, otherwise it goes to $ERROR$; and similarly for `unlock()`. From the specification, we know it is important whether or not the lock is held, hence we start by considering the two predicates $LOCK = 1$ and $LOCK = 0$. This is not necessary: even if we start with the empty set of predicates, the algorithm discovers the above predicates via bogus counterexamples.

Build Reachability Tree

The first phase of the algorithm is the search phase shown in Figure 3.2. The algorithm constructs in depth-first order the reachability tree. We omit the predicate maps from the figure for clarity. The numbers labelling the nodes are the CFA locations corresponding to the nodes' location. The formulas labelling the nodes are the nodes' regions, represent what is known about the state of the program with respect to the set of predicates being tracked, after executing the node trace, *i.e.,* the commands from the root of the tree to the given node. Each node's region is obtained by computing the region corresponding to the states the program is in (overapproximated to the predicates corresponding the edge command), if from a state in the parent region, it executes the command labelling the parent-node edge. Hence, each node's region is an overapproximation of the set of states actually reachable by executing the node's trace. Furthermore, for each node we have a finite set of predicates we consider (in our case, $LOCK = 1$ and $LOCK = 0$), and we require that the reachable region be described as a boolean combination of these predicates.

We begin with the node that corresponds to location 1 in the CFA. The only information we have at this point is the assumption (or precondition) that the lock is not held: $LOCK = 0$. The edge from 1 to 2 is a branch that can always be taken (labeled by `[T]`), hence at 2 also we know $LOCK = 0$. To go from 2 to 3, we call `lock`, and set `old = new`. As our predicates contain no information about the variables `new` and `old`, all we can conclude is that at 4, $LOCK = 1$. Similarly going from 4 to 5 we model only what happens to `LOCK`, so due to the `unlock`, at 5 we know $LOCK = 0$. From 5 to 6 is a branch that can only be taken if `new == old`. We know nothing about `new` and `old`, hence they could be equal, and so we take the branch and again at 6, we have $LOCK = 0$, as nothing affects `LOCK` during that transition. At 6 we see that we call `unlock` with the lock not held (as $LOCK = 0$), and hence we reach an *error node*.

Counterexample Analysis

When we hit an error node in the search tree, we check if the error node's trace is a genuine counterexample trace, *i.e.,* is a path that the program can actually follow and thus hit an error state, or results from the abstraction being too imprecise. For each node along the

path, the node's *error trace* is the sequence of commands labelling the suffix of the path from the node to the error node. To check if the error node's trace is genuine, we check if for each node along the error node's path, there is an actual state in the states corresponding to the node's region, that can execute the node's error trace and go to an error state.

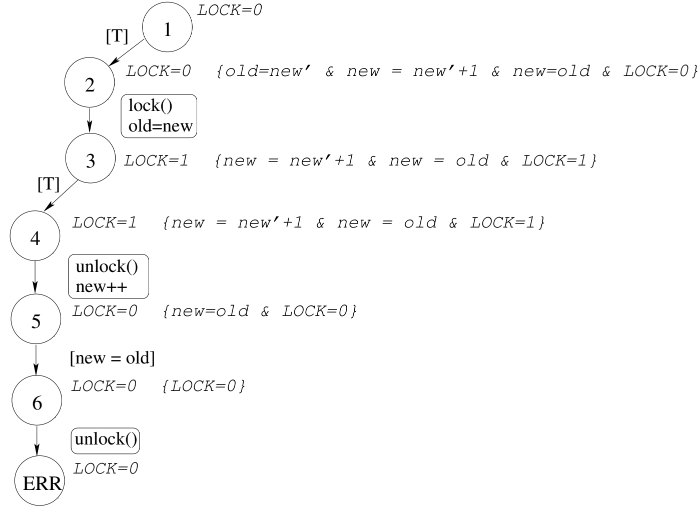

Figure 3.3: Counterexample Analysis

Figure 3.3 shows the result of this phase. In the figure, for each tree node, the formula in the curly braces, called the *bad region*, represents the set of states that the program can end up in, if from a state in the node's region (from the forward search phase), it executes the node's error trace. In other words, the formula is the strongest postcondition [Dij76] of the node's region with respect to node's error trace. We go backwards from the error node, trying to find the first node in the tree where the bad region (of the node) becomes empty. Upon finding such a node, conclude that it is *not* possible to reach the error via the given trace. That node becomes the *pivot node*, and we shall refine the abstraction from that node onwards.

Consider the bad regions labeling the nodes in Figure 3.3. Upon executing $\sigma_6 =$ `unlock`() from a state in location 6 where $LOCK = 0$, the program goes to location $ERROR$ and the variables satisfy $LOCK = 0$, which therefore, is the bad region of 6. Similarly, the bad region of 5 is the strongest postcondition of $LOCK = 0$ w.r.t. the trace

$\sigma_5 = \texttt{assume}[new = old]; \sigma_6$ which is $new = old \wedge LOCK = 0$. The former condition arises as the branch corresponding to the assume was taken, the latter arises because the program began in that state (at node 5), and the call `unlock`(), leaves `LOCK` unchanged when called with $LOCK = 0$. The bad region of 4 is the strongest postcondition of $LOCK = 1$ w.r.t. the trace $\sigma_4 = \texttt{unlock}(); \texttt{new}++; \sigma_5$, which is:

$$\exists new'.new = new' + 1 \wedge new = old \wedge LOCK = 0$$

The variable new' refers to the (stale) value of `new` at 4; the value of `new` after executing σ_4 is one greater than this stale value, and is equal to the value of *old* at the end (due to the assume). The bad region of 3 is the strongest postcondition of $LOCK = 0$ w.r.t. $\sigma_3 = \texttt{assume}[\texttt{T}]; \sigma_4$, which is the same as the bad region of 4, as `assume`[`T`] doesn't affect the state. Finally the bad region of 2 is the strongest postcondition of $LOCK = 0$ w.r.t. $\sigma_2 = \texttt{lock}(); \texttt{old} := \texttt{new}; \sigma_3$, which is:

$$\exists new'.old = new' \wedge new = new' + 1 \wedge new = old \wedge LOCK = 0$$

It is easy to check that this region is empty, *i.e.,* if the program begins at location 2 in a state satisfying $LOCK = 0$, there is no way that it can execute σ_2 and thus go to *ERROR*. In fact, the path from the node labeled 2 to the error node is the smallest infeasible suffix of the counterexample (which is the entire path from the root to the error node). Thus the node labeled 2 is the pivot node.

To do the emptiness check at each point we ask a theorem prover if the formula corresponding to the bad region is satisfiable. By partitioning the *proof of unsatisfiability* of this formula, we learn that the predicate $new = old$ is important (see Section 3.3 for details). The reason we hit an error node is that the abstraction is too imprecise, and tracking this new predicate will enable us to rule out this particular infeasible counterexample path. We add the new predicate $new = old$ to the map for the commands labelling the edges from 2 to 3, 3 to 4, and 4 to 5 in the subtree of the pivot node, and thus refine the abstraction from the pivot node onwards.

If our analysis had gone back all the way to the root without the theorem prover reporting unsatisfiability at any point, then the path from the root to the error node would

infact be a real error, *i.e.,* a counterexample demonstrating that the program violated the specification.

Resume Search with new predicates

We continue the search phase, searching from the pivot node onwards. This time, we track also the predicate $new = old$, and the resulting search tree can be seen in Figure 3.4.

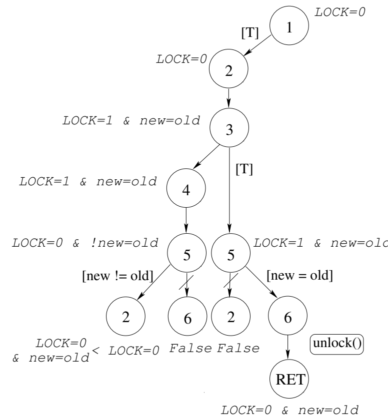

Figure 3.4: Search with new Predicate

Notice that we can stop the search at the leaf labeled 2, as the states satisfying that node's region $LOCK = 0 \wedge new = old$ are a subset of those satisfying $LOCK = 0$, hence any error found from this point on would have been found by exploring from the ancestor labeled 2. We call such nodes, whose regions are contained in the regions of ancestor nodes in the tree, *covered*. They correspond to fixpoints, and this is how loops are handled automatically. Whenever we see a covered node, we backtrack and search along some other branch in depth-first order.

Also, the region of the leaf labeled 6 is empty, as that node could be reached only if at 5 `new` equalled `old`, but in node 5, we know that $new \neq old$ (as this time we are tracking the relationship of `new` and `old`). Thus we do not search from that node any more, but backtrack to the node labeled 3 and follow its other branch. The region of Leaf 2 is empty as that branch is taken only when `new` is not equal to `old`. Moreover, the error node is not

reachable from the node labeled 6, as that node's region stipulates that the lock is held, and hence the call to `unlock` is safe. Thus we conclude that no error node is reachable in the entire left subtree.

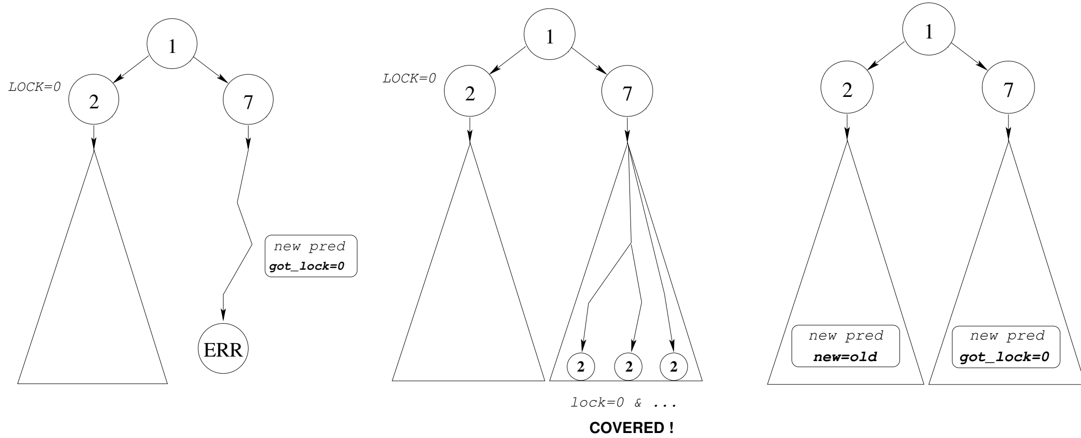

Figure 3.5: (a) Building Right Subtree (b) Leaves covered (c) Different abstractions

In Figure 3.5(a) we search the right subtree of the root in a similar fashion. We discover a bogus counterexample that gives us the predicate $got_lock = 0$. Considering this additional predicate is enough to rule out all error traces in the right subtree. Note that in the right subtree there are leaves with label 2, as control always flows to that point in the CFA. However they are all covered by the root node of the left subtree (Figure 3.5(b)), as the regions of each of these leaf nodes is contained in the region of the root of the left subtree (at all those nodes, the lock is not held), and hence, these leaves are covered. Hence, we need not continue to search any further, and conclude that no error node is reachable.

Savings

We have achieved two savings. First, each part of the state space is refined only as much as required; in particular we have different abstractions for the two subtrees (see Figure 3.5(c)). Second, we explore only the portion of the state space that is required in order to prove correctness, and do not throw away the work done earlier. For example, when we hit an error in the right subtree, we refine only that part of the search tree, keeping intact and using in the proof the left subtree, as we already know there is no error in that part of the

state space. In the following sections, we make this intuitive algorithm precise.

3.2 Symbolic Reachability with Refinement

We present two algorithms for computing overapproximations of the reachable state space of a labeled transition system. The first algorithm is nondeterministic. The second algorithm, which is the Lazy Abstraction algorithm, resolves the nondeterminism in the first one to ensure that the overapproximation does not contain any error states, if in fact no error state is reachable in the system. These algorithms do not terminate in general (so we should call them "semi-algorithms," but we keep the term algorithm for simplicity). Sufficient conditions ensuring termination will be discussed in Section 3.4. Our algorithms build an overapproximation of the reachable states of the LTS, by constructing an appropriate reachability tree.

Reachability Trees. Given an LTS $\mathcal{S} = (X, \Sigma, \rightsquigarrow, S_0)$, and a symbolic region structure $\mathcal{R} = (R, \bot, \sqcup, \sqcap, \mathsf{post}, [\![\cdot]\!])$ for $\mathcal{S}$, a *Reachability Tree* for $\mathcal{R}$ is a rooted directed tree, $T = (V, E, \mathtt{n}_0)$ where V is a set of nodes labelled with regions from R, $\mathtt{n}_0 \in V$ is the root node of the tree, and, E is a set of edges labelled with elements of Σ. We write $\mathtt{n}{:}r$ for the node $\mathtt{n} \in V$ labelled with region $r \in R$. We write $\mathtt{n}\xrightarrow{l}\mathtt{n}'$ for the edge from $\mathtt{n}$ to $\mathtt{n}'$ labelled l, and call $\mathtt{n}'$ the l-successor of $\mathtt{n}$. A node is called a *leaf* if it has no successors, otherwise it is called an *internal* node. A node $\mathtt{n} : r$ is *covered* if there exist internal nodes $\{\mathtt{n}_1 : r_1, \dots, \mathtt{n}_k : r_k\}$ such that $r \sqsubseteq \sqcup\{r_1, \dots, r_k\}$. A reachability tree is *complete* if:

1. Every edge $\mathtt{n}{:}r\xrightarrow{l}\mathtt{n}'{:}r'$ in the tree, is such that $\mathsf{post}.r.l \sqsubseteq r'$,

2. Every internal node, has an l-son for each l in Σ, and,

3. Every leaf node is covered.

A *partial* reachability tree is a pair (T, F) where F is a subset of the leaf nodes of T, and we require in place of (3) above, the condition (3') Every leaf node in $T \setminus F$ is covered.

The next theorem states that the union of the regions labelling the nodes of a reachability tree is an overaprroximation of the reachable states of the LTS.

Theorem 1 [Reachability Trees] *If $T = (V, E, \mathtt{n}_0)$ is a complete reachability tree for $\mathcal{R}$, where $\mathtt{n}_0 : r_0$, then:* $\mathsf{Reach}.\mathcal{S}.([\![r_0]\!]) \subseteq [\![\sqcup\{r \mid \mathtt{n}:r$ *is an internal node of* $T\}]\!]$.

Proof. Recall that $\mathsf{Reach}.\mathcal{S}.([\![r_0]\!])$ is the set $\{s \mid s_0 \overset{\sigma}{\rightsquigarrow} s$ for some $s_0 \in S_0, \sigma \in \Sigma^*\}$. We shall show, by induction on the length of σ, that if $s_0 \overset{\sigma}{\rightsquigarrow} s$ for some $s_0 \in [\![r_0]\!]$, and $\sigma \in \Sigma^*$, then there exists some internal $\mathtt{n}:r$ in T, such that $s \in [\![r]\!]$, and from this, the theorem follows. For the base case, $|\sigma| = 0$, the node is $\mathtt{n}_0 : r_0$. Suppose the induction hypothesis holds for all traces of length n. Consider a trace $\sigma' = \sigma \cdot l$ of length $n+1$, and s that $s_0 \overset{\sigma'}{\rightsquigarrow} s'$ for some $s_0 \in [\![r_0]\!]$. By the definition of $\rightsquigarrow$, there exists some s such that $s_0 \overset{\sigma}{\rightsquigarrow} s \overset{l}{\rightsquigarrow} s'$, and by the IH there exists an internal $\mathtt{n} : r$ such that $s \in [\![r]\!]$. As the tree is complete, $\mathtt{n} : r$ must have a l-son $\mathtt{n}' : r'$ such that $\mathsf{post}.r.l \sqsubseteq r'$. From $s \overset{l}{\rightsquigarrow} s'$ and $s \in [\![r]\!]$, we know that $s' \in r'$. If $\mathtt{n}'$ is an internal node we are done; if not, it is covered by a set of internal nodes $\{\mathtt{n}_1 : r_1, \ldots, \mathtt{n}_k : r_k\}$, such that for some $\mathtt{n}_i : r_i$ we have $s' \in r_i$. □

A reachability tree $T = (V, E, \mathtt{n}_0)$ for $\mathcal{R}$ is *safe* w.r.t. $\mathcal{E}$ from r_0 if T is a complete reachability tree such that (1) the root node $\mathtt{n}_0 : r_0$, and, (2) for every $\mathtt{n} : r$ in the tree, we have $r \sqcap \mathcal{E} \equiv \bot$. From Theorem 1, the following is immediate:

Proposition 2 [Safe Reachability Trees] *For a region structure $\mathcal{R}$ for LTS $\mathcal{S}$, and two regions r_0, $\mathcal{E}$, if $\mathcal{S}.S_0 \subseteq [\![r_0]\!]$ and there exists a reachability tree T for $\mathcal{R}$ that is safe w.r.t. $\mathcal{E}$ from r_0, then $\mathcal{S}$ is safe w.r.t. $[\![\mathcal{E}]\!]$.*

3.2.1 Reachability with refinement

The first algorithm we present is the algorithm SymbReachRefine (Algorithm 1). This is a standard symbolic forward-search algorithm, but with two extra features:

1. the ability to refine some path in the tree under construction (in order to compute a more precise abstraction of some part of the system), and

2. the ability to drop a subtree of the tree under construction (in order to recompute the subtree with more precision).

A classical symbolic forward-search algorithm computes a reachability tree, with nodes partitioned into internal (*Intl*) and leaf (*Leaf*). starting from an initial region r_0. The

Algorithm 1 $\mathsf{SymbReachRefine}(\mathcal{A}, r_0)$

Input: a region structure $\mathcal{R} = (R, \bot, \sqcup, \sqcap, \mathsf{post}, [\![\cdot]\!])$, an abstraction structure $\mathcal{A} = (\mathcal{R}, \widehat{\mathsf{post}}, \trianglelefteq)$, and an initial region $r_0 \in R$.

Output: A region r s.t. $\mathsf{Reach}.([\![r_0]\!]) \subseteq [\![r]\!]$

```
1:  L := {n0 : r0}, Intl := ∅                                              {Initialize}
2:  while L ≠ ∅ do
3:    Pick and remove n : r from L                          {process next worklist element}
4:    if (r ⊑ ⊔{r' | n' : r' ∈ Intl }) then
5:      Leaf := Leaf ∪ {n}                                                  {covered node}
6:    else
7:      if (∗) then
8:        Pick some path n' : r' —σ→ n : r in the tree where σ ∈ Σ*                {refine}
9:        relabel n' by a region w such that w ⊴ r' and w ≡ r'
10:       if (KeepSubtree.n') then
11:         L := L ∪ {n}
12:         for each n'' such that n' —σ'→ n'' → n do
13:           relabel n'' by post^.w.σ'                                        {refine path}
14:       else
15:         L := L ∪ {n'}
16:         remove subtree of n' from tree,L,Intl,Leaf                      {refine subtree}
17:       for each n'' ∈ Leaf labelled after n' do
18:         remove n'' from Leaf, add it to L                  {to guarantee correctness}
19:     else
20:       Intl := Intl ∪ {n}                                              {build reach tree}
21:       for each label l ∈ Σ do
22:         r' := post^.r.l
23:         construct an l-son n' : r' of n
24:         L := L ∪ {n'}
25: return the region ⊔{r | n : r ∈ Intl }
```

algorithm maintains a worklist L of nodes to be explored, and iteratively processes nodes until the worklist becomes empty. To process a node $\mathtt{n}$ from the worklist, we remove it from the worklist and check whether the node's reachable region is covered by the already explored state space(line 4).

If the node is covered we add it to the set of leaf nodes, and repeat with the next element of the worklist(line 5). If the node is not covered then we non-deterministically choose (line 7) to either refine the states at the node (lines 8–18), or to construct the sons of the node *i.e.,* compute the states that are reachable in one step from that node (lines 20–24).

If we choose to refine, then we pick some ancestor node $\mathtt{n}'$ of the node being processed, and relabel it with a more precise version of its region (lines 8–9). We then non-deterministically choose, via the function $\mathsf{KeepSubtree}$, to either just refine the nodes along the path from $\mathtt{n}'$ to $\mathtt{n}$, by computing the abstract post from the more precise region along the path (lines 12–13), or to delete and recompute, and hence, refine, the entire subtree rooted at $\mathtt{n}'$ (lines 15–16). In either case, some leaves that were previously covered, may become uncovered, as the internal nodes covering them have smaller regions due to refinement, and so we add these leaves to the worklist for processing (lines 17–18). To find which these leaves are, we attach with every node, a *time stamp* indicating the time when the node was labelled last; these time stamps linearly order all the nodes in the tree. If a leaf was labelled *after* some internal node, then refining the internal node may cause the leaf to become uncovered.

If instead, we chose to construct the sons, then for each label l, the l-sons are constructed using the $\widehat{\mathsf{post}}$ operation and added to the worklist.

When the worklist is empty, the algorithm returns the union of all the regions of the internal nodes of the tree. It can be shown, that at this point, the tree is a complete reachability tree, and so the returned region is an overapproximation of the states reachable from the initial region r_0. To see the above, notice that the following invariants hold of the partially constructed tree, at the start of each iteration of the the main loop.

Invariant 1. Every edge $\mathtt{n}:r \xrightarrow{l} \mathtt{n}':r'$ in the tree is such that: $\mathsf{post}.r.l \sqsubseteq \widehat{\mathsf{post}}.r.l = r'$.

Invariant 2. The nodes of the tree are partitioned into the sets:

- *Intl:* The nodes in *Intl* are internal nodes of the partial reachability tree, with an l-son, for each l in Σ.
- *Leaf:* The nodes in *Leaf* are covered leaf nodes.
- *L:* The nodes in L are the worklist nodes, *i.e.,* leaf nodes that are not covered, and which need to be processed so that either they are refined, and become covered, or they become internal nodes.

From the above invariants, it follows that if the worklist is empty, then the the nodes in $\mathit{Leaf} \cup \mathit{Intl}$ are a complete reachability tree.

The goal of this generic symbolic reachability algorithm is to point out sufficient conditions for correctness. Indeed, once this algorithm is proved correct, any more detailed implementation, *e.g.* replacing the nondeterministic $*$ and fixing a choice of the path to refine must necessarily be correct. The correctness of the SymbReachRefine algorithm is expressed by the following theorem, which follows from the fact that if SymbReachRefine.$(\mathcal{A}, r_0)$ terminates, then the tree rooted at $\mathtt{n}_0 : r_0$ is a complete reachability tree. Note that the correctness does not depend on the order in which the state space is explored (*e.g.* depth-first or breadth-first).

Theorem 2 [Correctness] *Let $\mathcal{A}$ be an abstraction structure for a labeled transition system $\mathcal{S}$. For every initial region r_0, and every terminating execution of the algorithm* SymbReachRefine.$(\mathcal{A}, r_0)$*, the tree rooted at $\mathtt{n}_0$ is a complete reachability tree and hence:* Reach.$\mathcal{S}.([\![r_0]\!]) \subseteq [\![$SymbReachRefine.$(\mathcal{A}, r_0)]\!]$.

3.2.2 Counterexample-driven refinement

We now present our algorithm LazyAbstraction for the safety verification problem, *i.e.,* to decide, given an initial region r_0 and an error region $\mathcal{E}$, whether the latter is reachable from the former. To show that $\mathcal{E}$ is not reachable from r_0, Algorithm LazyAbstraction tries to build a reachability tree that is safe w.r.t. $\mathcal{E}$ from r_0. If it can find such a tree then we know that the LTS is safe. The algorithm LazyAbstraction is an implementation of the previous algorithm where the nondeterministic choice for refinement is done using traces corresponding to bogus counterexamples, the more precise (*i.e.,* refined) region being

Algorithm 2 $\mathsf{LazyAbstraction}.(\mathcal{A}, \mathsf{Refine}, r_0, \mathcal{E})$

Input: a region structure $\mathcal{R} = (R, \bot, \sqcup, \sqcap, \mathsf{post}, [\![\cdot]\!])$ for LTS $\mathcal{S}$, an abstraction structure $\mathcal{A} = (\mathcal{R}, \widehat{\mathsf{post}}, \trianglelefteq)$, a refine operator Refine for $\mathcal{A}$, an initial region $r_0 \in R$, s.t. $[\![r_0]\!] = \mathcal{S}.S_0$ and an error region $\mathcal{E} \in R$.

Output: Either $\textsc{Safe}(r)$, where $\mathsf{Reach}.\mathcal{S}.([\![r_0]\!]) \subseteq [\![r]\!]$ and $r \sqcap \mathcal{E} \equiv \bot$, or $\textsc{Unsafe}(\sigma)$ where σ is a counterexample to $\mathcal{E}$ from r_0.

1: $L := \{\mathtt{n}_0 : r_0\}$, *Intl* $:= \emptyset$ {Initialize}
2: **while** $L \neq \emptyset$ **do**
3: Pick and remove $\mathtt{n}:r$ from L {process next worklist element}
4: **if** $(r \sqsubseteq \sqcup\{r' \mid \mathtt{n}':r' \in \mathit{Intl}\})$ **then**
5: $\mathit{Leaf} := \mathit{Leaf} \cup \{\mathtt{n}\}$ {covered node}
6: **else**
7: **if** $r \sqcap \mathcal{E} \not\sqsubseteq \bot$ **then**
8: {abstract counterexample found}
9: **if** there is a $\mathtt{n}':r' \xrightarrow{\sigma} \mathtt{n}:r$ s.t. $\mathsf{post}.r'.\sigma \sqcap \mathcal{E} \equiv \bot$ **then**
10: relabel $\mathtt{n}'$ by $w = \mathsf{Refine}.(r', l\sigma, \mathcal{E})$ {refine using bogus counterexample σ}
11: **if** $\mathsf{KeepSubtree}.\mathtt{n}'$ **then**
12: $L := L \cup \{\mathtt{n}\}$
13: **for all** $\mathtt{n}''$ such that $\mathtt{n}' \xrightarrow{\sigma'} \mathtt{n}'' \rightarrow \mathtt{n}$
14: relabel $\mathtt{n}''$ by $\widehat{\mathsf{post}}.w.\sigma'$ {refine path}
15: **else**
16: $L := L \cup \{\mathtt{n}'\}$
17: remove subtree of $\mathtt{n}'$ from tree,L,*Intl*,*Leaf* {refine subtree}
18: **for all** $\mathtt{n}'' \in \mathit{Leaf}$ labelled *after* $\mathtt{n}'$ **do**
19: remove $\mathtt{n}''$ from *Leaf*, add it to L {to guarantee correctness}
20: **else**
21: let σ be the trace such that $\mathtt{n}_0 \xrightarrow{\sigma} \mathtt{n}$ {σ is a genuine counterexample}
22: **return** $\textsc{Unsafe}(\sigma)$
23: **else**
24: $\mathit{Intl} := \mathit{Intl} \cup \{\mathtt{n}\}$ {build reach tree}
25: **for all** labels $l \in \Sigma$ **do**
26: $r' := \widehat{\mathsf{post}}.r.l$
27: construct an l-son $\mathtt{n}':r'$ of $\mathtt{n}$
28: $L := L \cup \{\mathtt{n}'\}$
29: **return** $\textsc{Safe}(\sqcup\{r \mid \mathtt{n}:r \in \mathit{Intl}\})$

obtained via a refine operator Refine.

The Refine Operator

The refinement step uses a *refine operator*, denoted Refine. Intuitively, a refine operator is needed when we have a a bogus abstract counterexample σ to $\mathcal{E}$ from some region r. We want $\mathsf{Refine}.(r, \sigma, \mathcal{E})$ to return a region w equivalent to r, but precise enough to rule out the the bogus error trace σ. Formally, a refine operator Refine for an abstraction structure $\mathcal{A}$ with region set R is a function $\mathsf{Refine}\colon\ R \times \Sigma^* \times R \to R$ such that for all regions $r, \mathcal{E} \in R$ and all $\sigma \in \Sigma^*$,

$$- \mathsf{Refine}.(r, \sigma, \mathcal{E}) \equiv r \text{ and } \mathsf{Refine}.(r, \sigma, \mathcal{E}) \trianglelefteq r, \text{ and}$$

$$- \text{if } \mathsf{post}.r.\sigma \sqcap \mathcal{E} \equiv \bot, \text{ then } \widehat{\mathsf{post}}.(\mathsf{Refine}(r, \sigma, \mathcal{E})).\sigma \sqcap \mathcal{E} \equiv \bot.$$

The second condition is not necessary for correctness of the algorithm $\mathsf{LazyAbstraction}$, but it will allow us to obtain a termination criterion in Section 3.4. In parsimonious predicate abstraction, a refine operator adds predicates for the various commands in the bogus counterexample trace in order to render the trace abstractly infeasible, but leaves the boolean formula characterizing the node's region unchanged. As the predicates are added locally, at any time there may be regions in the reachability tree with different predicate maps. We defer the discussion of a refine operator for imperative programs which enjoys the above properties to Section 3.3.

The Algorithm $\mathsf{LazyAbstraction}$

The algorithm $\mathsf{LazyAbstraction}$ builds reachability tree that is precise enough to demonstrate that no error state is reached from the initial states, or if that is not the case, then the algorithm returns a counterexample trace from the initial region to the error region.

The $\mathsf{LazyAbstraction}$ algorithm behaves exactly like a usual symbolic forward-search algorithm $\mathsf{SymbReachRefine}$, as long as every time a node is picked from the worklist, the node's region has an empty intersection with the error region. However, when it finds a node whose region contains an error state, it checks if the abstract counterexample trace corresponding to the path in the tree to the node is genuine or not. If the counterexample is genuine then $\mathsf{LazyAbstraction}$ returns it, reporting the system unsafe. If the counterexample

is bogus, it uses the refine operator on the trace to refine the abstraction, and continue. This is made precise in the algorithm LazyAbstraction (Algorithm 2).

As before, we use a worklist and iteratively build the reachability tree. When the algorithm processes a node $\mathtt{n}$ whose reachable region has a nonempty intersection with the error region (line 7), it checks if this is an actual error. For each node $\mathtt{n}'$ along the path from the root $\mathtt{n}_0$ to $\mathtt{n}$, let the *bad trace* of the node $\mathtt{n}'$ be the trace σ' labelling the path from $\mathtt{n}'$ to $\mathtt{n}$ in the tree. The *bad region* of $\mathtt{n}'$ is the set of error states that the system can be in after executing the bad trace from a state in the reachable region of $\mathtt{n}'$, *i.e.,* the bad region of $\mathtt{n}' : r'$ is $\mathsf{post}.r'.\sigma' \sqcap \mathcal{E}$.

The algorithm checks for every ancestor $\mathtt{n}'$ of $\mathtt{n}$ if the bad region of the ancestor is empty (line 9). If so, then $\mathtt{n}'$ is a *pivot node*, and the bad trace from $\mathtt{n}' : r'$ is a bogus counterexample to $\mathcal{E}$ from r' that we shall use it to refine our abstraction (line 10). If there is no pivot node, in particular if the root node $\mathtt{n}_0$ is not a pivot, then its bad trace is a genuine counterexample to $\mathcal{E}$ from r_0, and it is returned (lines 20–22).

The algorithm refines the search from the pivot node on, by relabeling the pivot with a more precise version of its region using the refine operator Refine. If it is worth keeping the whole subtree of the pivot node (which is determined by the KeepSubtree heuristic), then it refines the path from the pivot node to the currently processed node $\mathtt{n}$ (lines 12–14); otherwise it removes the entire subtree rooted at the pivot node and adds the pivot to the worklist (lines 16–17). The rest of the algorithm is identical to SymbReachRefine.

The KeepSubtree function determines whether the subtree rooted at the pivot node is discarded in the refinement process. It does not affect correctness, but may affect termination. For efficiency, we want to keep as much computation as possible (to avoid repeating the same work), and hence we would like to keep subtrees. But there may be coarse nodes in the subtrees that can cause the algorithm to go into infinite "refinement loops." The use of a KeepSubtree function allows us to experiment with different strategies.

Example 7 [Refinement Loops] Consider the CFA given in Figure 3.6(a). The result of the first search phase is given in (b). The second node labeled 2 is covered by its parent. Along the other branch the search hits an error node. Figure (c) shows how the error path is refined, by adding the support predicate $x = 0$: at the parent 2 node, the reachable

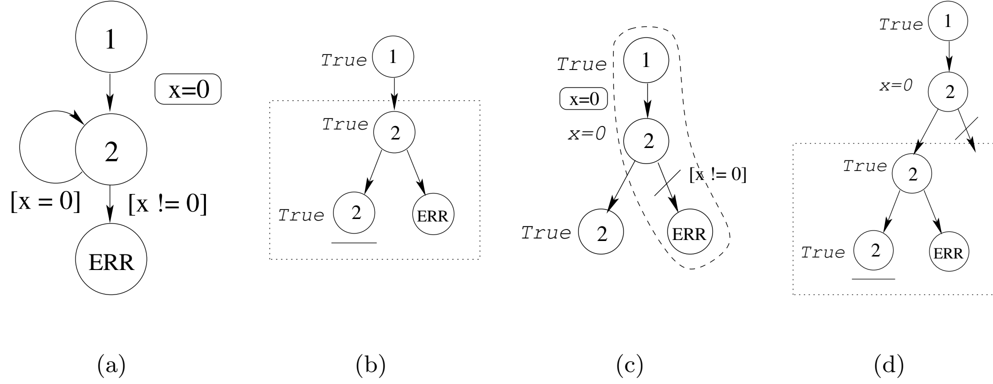

Figure 3.6: Refinement loops

region is now $x = 0$, so the branch to the error node is ruled out. The child 2 node is now no longer covered, so it is unmarked and the search resumes from that node. In (d) we see that the search results in exactly the same subtree we had in (b), hence, the refine-search process will repeat forever. If, instead, we delete the entire subtree below the pivot node (*i.e.,* the node labeled 1), unmark the pivot node, and start over searching with the new predicate $x = 0$ from the pivot node onwards, then the algorithm terminates. □

If the $\mathsf{LazyAbstraction}.(\mathcal{A}, \mathsf{Refine}, r_0, \mathcal{E})$ terminates and returns $\textsc{Safe}(r)$, then, using Theorem 2, we can show that the tree rooted at $\mathtt{n}_0$ is a complete reachability tree, and the properties of LazyAbstraction ensure that the root is labelled with r_0 and every node has an empty intersection with $\mathcal{E}$ and hence, the tree is safe w.r.t. $\mathcal{E}$ from r_0. Hence, from Proposition 2, we conclude that $\mathcal{S}$ is safe w.r.t. $\mathcal{E}$ from r_0. If instead, it returns $\textsc{Unsafe}(\sigma)$, then σ is a counterexample to $\mathcal{E}$ from $[\![r_0]\!]$ and hence, $\mathcal{S}$ is not safe w.r.t. $\mathcal{E}$ from r_0. The correctness of the LazyAbstraction algorithm, which again does not depend on the order in which the state space is explored, is expressed by the following theorem.

Theorem 3 [Correctness] *Let $\mathcal{A}$ be an abstraction structure for an LTS $\mathcal{S}$, and let* Refine *be a refine operator for $\mathcal{A}$. For every initial region $[\![r_0]\!] = \mathcal{S}.S_0$, and error region $\mathcal{E}$, if* $\mathsf{LazyAbstraction}.(\mathcal{A}, \mathsf{Refine}, r_0, \mathcal{E})$ *returns:*

1. $\textsc{Safe}(\cdot)$*, then $\mathcal{S}$ is safe w.r.t. $\mathcal{E}$,*

2. $\textsc{Unsafe}(\cdot)$*, then $\mathcal{S}$ is not safe w.r.t. $\mathcal{E}$.*

	(a)	(b)	(c)
1:	$x := ctr;$	$\langle x, 1\rangle = \langle ctr, 0\rangle$	$x = ctr$
2:	$ctr := ctr + 1;$	$\langle ctr, 1\rangle = \langle ctr, 0\rangle + 1$	$x = ctr - 1$
3:	$y := ctr;$	$\langle y, 2\rangle = \langle ctr, 1\rangle$	$x = y - 1$
4:	`assume` $(x = m);$	$\langle x, 1\rangle = \langle m, 0\rangle$	$y = m + 1$
5:	`assume` $(y \neq m + 1);$	$\langle y, 2\rangle = \langle m, 0\rangle + 1$	

Figure 3.7: (a) Infeasible trace (b) Constraints (c) Predicates.

We mention two optimizations of Algorithm 2. First, if nodes from the worklist are picked in depth-first search order then for every ancestor `n` of the latest labelled node, the marked nodes in the subtree of `n` are precisely the nodes that were labelled *after* `n` was marked. Hence the **for**-loop of lines 17–18 can be limited to the leaves in the subtree of `n`. Second, in order to implement efficiently the covering test of line 4, we use a variable c to collect the union of reachable regions of internal nodes as we go along. We can then replace the covering test of line 4 by the test $r \sqsubseteq c$, and the return statement of line 27 by **return** c. To update c, we add the statement $c \leftarrow c \sqcup r$ after line 24, and we recompute c after line 17.

3.3 A Refine operator for Imperative Programs

We now present a Refine operator for imperative programs. Recall that the task at hand is the following: given a bogus abstract counterexample trace, we wish to wish to find a more precise abstraction in which the trace becomes abstractly infeasible.

Our approach is the following: we shall encode the trace as a formula called the trace formula, which is unsatisfiable iff the trace is (concretely) infeasible. We shall then partition the *proof of unsatisfiability* of the trace formula, to obtain a new set of support predicates. The support predicates are a map from commands to sets of predicates. The problem, then, is (i) to extract predicates from the proof, together with (ii) information *where* to use each predicate, such that the refined abstraction no longer contains the infeasible trace. We shall first give an intuitive overview of the technique, using the infeasible trace shown in Figure 3.7(a), and then describe it formally.

3.3.1 Overview

Symbolic Simulation. One possible approach to designing a refine operator is to symbolically simulate the trace until an inconsistent state is reached; such inconsistencies can be detected by decision procedures [BPS00]. A dependency analysis can be used to compute which events in the trace cause the inconsistency, and this set of events can then be heuristically minimized to obtain a suitable set of predicates [BR02a; CCGS03]. There are two problems with this approach. First, the inconsistency may depend upon "old" values of program variables, *e.g.* in the trace shown, such an analysis would use facts like x equals "the value of *ctr* at line 1," and that the "current" value of *ctr* is one more than the "value at line 1." In general there may be many such old values, and not only must one use heuristics to deduce which ones to keep, a problem complicated by the presence of pointers and procedures, but one must also modify the program appropriately in order to explicitly name these old values. Intuitively, however, since the program itself does not remember "old" values of variables, and yet cannot follow the path, it must be possible to track relationships between "live" values of variables only, and still show infeasibility. Second, this approach yields no information about *where* a predicate is useful.

We now demonstrate our technique on the trace of Figure 3.7(a). First, we build a *trace formula* (TF) which is satisfiable iff the trace is feasible. The TF φ is a conjunction of constraints, one per instruction in the trace. In Figure 3.7(b), the constraint for each instruction is shown on the right of the instruction. Each term $\langle\cdot,\cdot\rangle$ denotes a special constant which represents the value of some variable at some point in the trace, *e.g.* $\langle ctr, 1\rangle$ represents the value of *ctr* after the first two instructions. The constraints are essentially the strongest postconditions, where we give new names to variables upon assignment [CFR+91; FS01]. Thus, for the assignment in line 1, we generate the constraint $\langle x, 1\rangle = \langle ctr, 0\rangle$, where $\langle x, 1\rangle$ is a new name for the value of x after the assignment, and $\langle ctr, 0\rangle$ is the name for *ctr* at that point. Notice that the "latest" name of a variable is used when the variable appears in an expression on the right. Also note that the conjunction φ of all constraints is unsatisfiable.

To compute the new set of predicates, we could simply take all atomic predicates that

occur in the constraints, rename the constants to corresponding program variables, create new names ("symbolic variables") for "old" values of a variable *e.g.* for $\langle ctr, 1\rangle = \langle ctr, 0\rangle + 1$ create a new name that denotes the value of ctr at the previous instruction, and add these names as new variables to the program. However, such a support set is often too large, and in practice [BR02a; HJMS02] one must use heuristics to minimize the sets of predicates and symbolic variables by using a minimally infeasible subset of the constraints.

Craig interpolation. Given a pair (φ^-, φ^+) of formulas, an *interpolant* for (φ^-, φ^+) is a formula ψ such that (i) φ^- implies ψ, (ii) $\psi \wedge \varphi^+$ is unsatisfiable, and (iii) the variables of ψ are common to both φ^- and φ^+. If $\varphi^- \wedge \varphi^+$ is unsatisfiable, then an interpolant always exists [Cra57], and can be computed from a proof of unsatisfiability of $\varphi^- \wedge \varphi^+$. If $\mathcal{P}$ is a proof of unsatisfiability of $\varphi^- \wedge \varphi^+$, then we write $\mathsf{ITP}.(\varphi^-, \varphi^+).(\mathcal{P})$ for the extracted interpolant for (φ^-, φ^+).

In our example, suppose that $\mathcal{P}$ is a proof of unsatisfiability for the TF φ. Now consider the partition of φ into φ_2^-, the conjunction of the first two constraints ($\langle x, 1\rangle = \langle ctr, 0\rangle \wedge \langle ctr, 1\rangle = \langle ctr, 0\rangle + 1$), and φ_2^+, the conjunction of the last three constraints ($\langle y, 2\rangle = \langle ctr, 1\rangle \wedge \langle x, 1\rangle = \langle m, 0\rangle \wedge \langle y, 2\rangle = \langle m, 0\rangle + 1$). The symbols common to φ_2^- and φ_2^+ are $\langle x, 1\rangle$ and $\langle ctr, 1\rangle$; they denote, respectively, the values of x and ctr after the first two operations of the trace. The interpolant $\mathsf{ITP}.(\varphi_2^-, \varphi_2^+).(\mathcal{P})$ is $\psi_2 = (\langle x, 1\rangle = \langle ctr, 1\rangle - 1)$. Let $\hat{\psi}_2$ be the formula obtained from ψ_2 by replacing each constant with the corresponding program variable, *i.e.,* $\hat{\psi}_2 = (x = ctr - 1)$. Since ψ_2 is an interpolant, φ_2^- implies ψ_2, and so $x = ctr - 1$ is an overapproximation of the set of states that are reachable after the first two instructions (as the common constants denote the values of the variables after the first two instructions). Moreover, by virtue of being an interpolant, $\psi_2 \wedge \varphi_2^+$ is unsatisfiable, meaning that from no state satisfying $\hat{\psi}_2$ can one execute the remaining three instructions, *i.e.,* the suffix of the trace is infeasible for all states with $x = ctr - 1$. If we partition the TF φ in this way at each point $i = 1, \ldots, 4$ of the trace, then we obtain from $\mathcal{P}$ four interpolants $\psi_i = \mathsf{ITP}.(\varphi_i^-, \varphi_i^+).(\mathcal{P})$, where φ_i^- is the conjunction of the first i constraints of ϕ, and φ_i^+ is the conjunction of the remaining constraints. Upon renaming the constants, we arrive at the formulas $\hat{\psi}_i$, which are shown in the rightmost column of Figure 3.7. We collect the atomic predicates that occur in the formulas $\hat{\psi}_i$ in the predicate map Π by letting Π map

op_i to the predicate $\hat{\psi}_i$, for $i = 1, \ldots, 4$.

We can prove that the trace is abstractly infeasible using the predicate map Π. Intuitively, for each point $i = 1, \ldots, 4$ of the trace, the formula $\hat{\psi}_i$ represents an overapproximation of the states s such that s is reachable after the first i instructions of the trace, and the remaining instructions are infeasible from s. From Proposition 3 of Section 3.3.2, it follows that $\mathsf{SP}.(\hat{\psi}_i).\mathsf{op}_{i+1}$ implies $\hat{\psi}_{i+1}$, for each i. For example, $\mathsf{SP}.(x = ctr - 1).(y := ctr)$ implies $x = y - 1$. Therefore, by adding all predicates from all ψ_i to Π, we have $\mathsf{SP}_\Pi.true.(\mathsf{op}_1 \cdot \ldots \cdot \mathsf{op}_i)$ implies $\hat{\psi}_i$. Note that, as the trace is infeasible, $\hat{\psi}_5 = \psi_5 = false$. Thus, $\mathsf{SP}_\Pi.true.(\mathsf{op}_1 \cdot \ldots \cdot \mathsf{op}_5)$ implies *false*, *i.e.,* the trace is abstractly infeasible (cf. Section 2.3).

Locality. The interpolants give us even more information. Consider the naive method of looking at just the TF. The predicates we get from it are such that we must track all of them all the time. If, for example, after the third instruction, we forget that x equals the "old" value of ctr, then the subsequent `assume[]` does not tell us that $y = m + 1$ (dropping the fact about x breaks a long chain of reasoning), thus making the trace abstractly feasible. In this example, heuristic minimization cannot rule out any predicates, so all predicates that occur in the proof of unsatisfiability of the TF must be used at all points in the trace. Using the interpolant method, we show that for infeasible traces of length n, the formula $\mathsf{SP}_{\hat{\psi}_n}.(\ldots(\mathsf{SP}_{\hat{\psi}_1}.true.\mathsf{op}_1)).\mathsf{op}_n$ is unsatisfiable (see Theorem 4 for a precise statement of this). Thus, at each point i in the trace, we need only to track the predicates in $\hat{\psi}_i$. For example, after executing the first instruction, all we need to know is $x = ctr$, after the second, all we need to know is $x = ctr - 1$, after the third, all we need to know is $x = y - 1$, and so on. This gives us a way to localize predicate usage. Thus, instead of a monolithic set of predicates all of which are relevant at all points of a trace, we can deduce a small set of predicates for each command of the trace.

3.3.2 Interpolants from Proofs

Consider formulas in the quantifier-free fragment of *FOL*. A sequent is of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas. The interpretation of $\Gamma \vdash \Delta$ is that the conjunction of the formulas in Γ entails the disjunction of the formulas in Δ.

$$\textsc{Hyp}\frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma$$

$$\textsc{Comb}\frac{\Gamma \vdash 0 \leq x \quad \Gamma \vdash 0 \leq y}{\Gamma \vdash 0 \leq c_1 x + c_2 y} c_{1,2} > 0$$

$$\textsc{Cong}\frac{\Gamma \vdash x_1 = y_1 \ \ldots \ \Gamma \vdash x_k = y_k}{\Gamma \vdash f^k(x_1, \ldots, x_k) = f^k(y_1, \ldots, y_k) 0} f^k \text{ is } k\text{-ary}$$

$$\textsc{Contra}\frac{\{\phi_1, \ldots, \phi_n\} \vdash 0 \leq c}{\Gamma \vdash \neg\phi_1, \ldots, \neg\phi_n}\ c < 0$$

$$\textsc{Res}\frac{\Gamma \vdash \{\phi\} \cup \Theta \quad \Gamma \vdash \{\neg\phi\} \cup \Theta'}{\Gamma \vdash \Theta \cup \Theta'}$$

Figure 3.8: Proof system.

We use a decision procedure[Nel81; BHJ+] that generates refutations for sets of clauses using the sequent proof system of Figure 3.8. In particular, all boolean reasoning is done by resolution. This system is complete for refutation of clause systems over the rationals. We obtain an incomplete system for the integers by systematically translating the literal $\neg(0 \leq x)$ to $0 \leq -1 - x$, which is valid for the integers.

An *interpolated sequent* is of the form $(\varphi^-, \varphi^+) \vdash \Delta\ [\psi]$, where φ^- and φ^+ are sets of clauses, Δ is a set of formulas, and ψ is a formula. This encodes the following three facts: (1) $A \vdash A'$, (2) $A', B \vdash \phi$, and, (3) Every variable occurring in A' also occurs in B or in ϕ. Note that if $(\varphi^-, \varphi^+) \vdash \mathit{false}[\psi]$, then ψ is an interpolant for (φ^-, φ^+).

There is a system of rules for corresponding to the rules of the proof system using which we can derive interpolated sequents that are sound, in that they are valid, and complete, in the sense that we can translate the derivation of any sequent $\varphi^- \cup \varphi^+ \vdash \Delta$ into the derivation of an interpolated sequent $(\varphi^-, \varphi^+) \vdash \Delta[\psi]$.

In particular, we can show that for every derivation $\mathcal{P}$ of a sequent $(\varphi^-, \varphi^+) \vdash \phi$ in our original proof system, there is a corresponding derivation $\mathcal{P}'$ of an interpolated sequent of the form $(\varphi^-, \varphi^+) \vdash \phi\ [\psi]$. We will refer to the interpolant ψ thus derived as $\mathsf{ITP}.(\varphi^-, \varphi^+).(\mathcal{P})$.

Using the same proof but partitioning the antecedent differently, we can obtain related interpolants. For example, we can show the following fact:

Proposition 3 *Let $\varphi^-, \varphi^+, \phi$ be formulas in FOL and $\mathcal{P}$ be a derivation of $\varphi^-, \varphi^+, \phi \vdash$ false. Then,* $\mathsf{ITP}.(\varphi^-, \phi \cup \varphi^+).(\mathcal{P}) \wedge \phi \Rightarrow \mathsf{ITP}.(\varphi^- \cup \phi, \varphi^+).(\mathcal{P})$.

The interpolant derivation rules have been proposed and studied by others and are beyond the scope of this dissertation. We refer the reader to the proof-theory literature [Kra97; Pud97] for the rules for boolean reasoning and linear arithmetic and to the work of Ken McMillan on combining the above with the theory of equality and uninterpreted functions [McM03; McM04], which was the inspiration for our work.

3.3.3 The Algorithm Refine

The refine operator is given an initial region r, a trace σ and an error region $\mathcal{E}$, such that σ is a bogus counterexample to $\mathcal{E}$ from r. It must return a region r' that is a more precise version of r such that σ is *not* an abstract counterexample to $\mathcal{E}$ from r'. In the case of imperative programs, the error region is a particular location $pc_{\mathcal{E}}$ in the program, *i.e.,* $\mathcal{E} = (pc_{\mathcal{E}}, (true, \cdot), \cdot)$. We assume, for clarity, that the starting region is the atomic region $(pc, (\varphi, \Pi), cs)$. In this case, the trace σ is a bogus counterexample because $\mathsf{SP}_{\Pi}.\varphi.\sigma$ is satisfiable, even though $\mathsf{SP}.\varphi.\sigma$ is not. It suffices to find a new support predicate set Π' such that $\mathsf{SP}_{\Pi'}.\varphi.\sigma$ is unsatisfiable, and then the refined region returned, *i.e.,* $\mathsf{Refine}.(r, \sigma, \mathcal{E})$ is just $(pc, (\varphi, \Pi'), cs)$.

Our method, described in Section 3.3.1, is made precise in Algorithm 3. First, we initialize Π' to be the empty map. Second, we use an operator Con, to build a *constraint map* Γ, a function that maps each point i of the trace σ to a *constraint* that corresponds to the ith operation of the trace. The conjunction of the constraints that are generated at all points of σ is the trace formula (TF) for σ, which is satisfiable iff the trace is feasible. Third, we find a proof $\mathcal{P}$ of the unsatisfiability of the TF. Such a proof exists as σ a bogus counterexample, and hence the TF is unsatisfiable. Fourth, for each point i in the trace, we *cut* the constraints into those from the first i commands (φ^-) and those from the remaining commands (φ^+). Using the proof $\mathcal{P}$ we compute the interpolant ψ for (φ^-, φ^+) and add the atomic predicates that occur in ψ after cleaning to the predicate map Π'. Finally, the

Algorithm 3 Refine

Input: A region $r = (pc, (\varphi, \Pi), cs)$, a trace σ, region $\mathcal{E} = (pc_{\mathcal{E}}, (true, \cdot), \cdot)$, s.t. σ is a bogus counterexample from r to $\mathcal{E}$.

Output: A region r', s.t. σ is not an abstract counterexample to $\mathcal{E}$ from r'.

$\Pi' := \emptyset$ {Initialize}
$(\cdot, \Gamma) := \mathsf{Con}.(\theta_0, \Gamma_0 :: \mathsf{IC}.\theta_0.\varphi)$ {Build Trace Formula}
$\mathcal{P} :=$ derivation of $\bigwedge_{1\le i\le n} \Gamma.i \vdash false$ {Find Proof of Unsatisfiability}
for $i := 1$ to $|\sigma|$ **do**
{Cut TF at each point}
  $\varphi^- := \bigwedge_{1\le j\le i} \Gamma.j$
  $\varphi^+ := \bigwedge_{i+1\le j\le n} \Gamma.j$
  $\psi := \mathsf{ITP}.(\varphi^-, \varphi^+).(\mathcal{P})$
  $\Pi'.(\sigma.i) := \mathsf{Atoms}.(\mathsf{Clean}.\psi)$ {Add predicates from interpolant at cut}
return $(pc, (true, \Pi \cup \Pi'), cs)$.

region returned is the input region with the new support predicates Π' added to the original support set. The correctness of this procedure is stated in Theorem 4.

Theorem 4 [Correctness of Refine] *For any trace σ and region $r = (pc, (\varphi, \Pi), cs)$, if* $\mathsf{SP}.\varphi.\sigma$ *is unsatisfiable, i.e., σ is infeasible from r then* $\mathsf{SP}_{\Pi'}.\varphi.\sigma$ *is unsatisfiable i.e., σ is abstractly infeasible from* $\mathsf{Refine}.(r, \sigma, \cdot) = (pc, (\varphi, \Pi'), cs)$.

Proof. *(Sketch)* Let $r' = \mathsf{Refine}.(r, \sigma, \cdot)$. It suffices to show that at the end of the Algorithm 3, the Π' is such that $\mathsf{SP}_{\Pi'}.\varphi.\sigma$ is unsatisfiable, as from this it follows that $\widehat{\mathsf{post}}.r'.\sigma \equiv \bot$), and hence σ is not an abstract counterexample to $\mathcal{E}$. Let σ_i denote the prefix of the first i operations of σ, namely $\sigma.1 \cdot \ldots \cdot \sigma.i$, and let ψ_i be the cleaned interpolant from the i-cut, namely $\mathsf{Clean}.\psi$, at the end of the ith iteration of the for loop. We can show, by induction on i, and using Proposition 3, that $\mathsf{SP}_{\Pi'}.\varphi.\sigma_i \Rightarrow \psi_i$. As the TF is unsatisfiable, $\psi_{|\sigma|}$ is *false*, and so $\mathsf{SP}_{\Pi'}.\varphi.\sigma \Rightarrow \psi_{|\sigma|} \Rightarrow false$ *i.e.,* $\mathsf{SP}_{\Pi'}.\varphi.\sigma$ is unsatisfiable. □

In particular, Theorem 4 states that our predicate discovery procedure Refine is *complete* in the sense that if the refined region r' returned is such that the trace is abstractly infeasible from it, and hence is not an abstract counterexample from r. Note that we lose precision by modelling operations like multiplication with uninterpreted functions, and the integers with rationals. In all these cases though, the "concrete" semantics, obtained via the SP operator, are themselves approximations. For the latter case, if the trace is deterministic, we never

l	$\mathsf{SP}.\varphi.l$	$\mathsf{Con}.(\theta,\Gamma).l$
$(\mathtt{f()}:\cdot)$	φ	$(\theta, \Gamma :: \mathit{true})$
$(\mathtt{return}:\cdot)$	“”	“”
$(\mathtt{x} := \mathtt{e}:\cdot)$	$\exists x'.\ (\varphi[x'/x] \wedge\ x = e[x'/x])$ where x' is a fresh variable	$(\theta', \Gamma :: \mathsf{Sub}.\theta'.(x = \mathsf{Sub}.\theta.e))$ where $\theta' = \mathsf{Upd}.\theta.\{x\}$
$(\mathtt{assume}[\mathtt{p}]:\cdot)$	$\varphi \wedge\ p$	$(\theta, \Gamma :: \mathsf{Sub}.\theta.p)$

Figure 3.9: Postconditions and Constraints for PI traces.

assign an arbitrary integer value to a variable, then we can assume that all integer variables are initialized to some default integer value, say 0. In this case all satisfying assignments of the SP of a trace will be integral even if the SP is interpreted over the rationals. Thus, if the trace is infeasible over the integers, our proof system can derive the unsatisfiability of the strongest postcondition.

We now fill in the details of the above algorithm 3, by showing how to generate the constraint map using the operators Con and IC, and by showing how Clean works. As before, we shall first discuss pointer-free programs, and then generalize to programs with pointers.

Constraints for Pointer-free Programs : PI

Lvalue Maps. An *lvalue map* is a function θ from $\mathsf{Lvals}.X$ to $\mathbb{N}$. The operator Upd: $(\mathsf{Lvals}.X \to \mathbb{N}) \to 2^{\mathsf{Lvals}.X} \to (\mathsf{Lvals}.X \to \mathbb{N})$ takes a map θ and a set of lvalues L, and returns a map θ' such that $\theta'.l = \theta.l$ if $l \notin L$, and $\theta'.l = i_l$ for a fresh integer i_l if $l \in L$. The function Sub takes an lvalue map θ and an lvalue l and returns $\langle l, \theta.l\rangle$. The function $\mathsf{Sub}.\theta$ is extended naturally to expressions and formulas. A *new* lvalue map is one whose range is disjoint from all other maps. We use lvalue maps to generate trace formulas (TF); at a point in the trace, if the map is θ, then the the pair $\langle l, \theta.l\rangle$ is a special constant that equals the value of l at that point in the trace. Whenever some lvalue l is updated, we update the map so that a fresh constant is used to denote the new value of l. For every such constant $c = \langle l, i\rangle$, let $\mathsf{Clean}.c = l$. The operator Clean can be naturally extended to expressions and formulas of *FOL*.

	$true$	
$(\texttt{assume}\ (b > 0) : pc_1)$;	$\langle b, 0\rangle > 0$	
$(c := 2 * b : pc_2)$;	$\langle c, 1\rangle = 2 * \langle b, 0\rangle$	
$(a := b : pc_3)$;	$\langle a, 2\rangle = \langle b, 0\rangle$	
$(a := a - 1 : pc_4)$;	$\langle a, 3\rangle = \langle a, 2\rangle - 1$	φ^-
$(\texttt{assume}\ (a < b) : pc_5)$;	$\langle a, 3\rangle < \langle b, 0\rangle$	φ^+
$(\texttt{assume}\ (a = c) : pc_6)$	$\langle a, 3\rangle = \langle c, 1\rangle$	

Figure 3.10: Cutting a PI trace.

Constraint Maps. A *constraint map* is map $\Gamma : \mathbb{N} \to FOL \cup \bot$. By Γ_0 we denote the map: $\lambda n.\bot$, *i.e.,* which returns $\bot$ everywhere. By $|\Gamma|$ we denote $\min\{n \mid \Gamma.n = \bot\}$. Given a formula φ in *FOL*, by $\Gamma :: \varphi$ we denote the map $\Gamma[|\Gamma| \mapsto \varphi]$. By $\bigwedge_n \Gamma$ we denote the formula $\bigwedge_{i \leq n} \Gamma.i$, where occurrences of $\bot$ are replaced with *true*.

Constraint Generation. The constraints are generated by the function Con, which takes a pair (θ, Γ) consisting of an lvalue map θ and a constraint map Γ, and a command $(pc : \mathsf{op})$, and returns a pair (θ', Γ') consisting of a new lvalue map and constraint map. The map is extended to traces as: $\mathsf{Con}.(\theta, \Gamma).\epsilon = (\theta, \Gamma)$ and $\mathsf{Con}.(\theta, \Gamma).l\sigma = \mathsf{Con}.(\mathsf{Con}.(\theta, \Gamma).l).\sigma$.

The function Con is defined in Figure 3.9, where we also repeat the definition of SP, as the generated constraints are a skolemized version of the strongest postcondition. We generate one constraint per command: If the operation is an assignment $\mathtt{x} := \mathtt{e}$, we first update the lvalue map so that a new constant denotes the value of x, and the constraint specifies that the new constant for x has the same value as the expression e (with appropriate constants substituted for program variables). If the operation is an assume operation $\texttt{assume}[\mathtt{p}]$, the constraint stipulates that the constants at that point satisfy the formula p. The initial constraint $\mathsf{IC}.\theta_0.\varphi$ is just $\mathsf{Sub}.\theta_0.\varphi$. For any formula φ in *FOL*, we can show, by induction over the length of σ, that $\mathsf{Sub}.\theta.(\mathsf{SP}.\varphi.\sigma)$ is equivalent (upto renaming of variables) to $\bigwedge_{|\sigma|} \Gamma$ where $(\theta, \Gamma) = \mathsf{Con}.(\theta_0, \Gamma_0 :: \mathsf{Sub}.\theta_0.\varphi).\sigma$. Here $\Gamma.i$ is the constraint for command $\sigma.i$. This gives the following propostion.

Proposition 4 [Equisatisfiability] *For a trace σ, and formula φ in FOL, let $(\cdot, \Gamma) = \mathsf{Con}.(\theta_0, \Gamma_0 :: \mathsf{IC}.\theta_0.\varphi).\sigma$. Then $\mathsf{SP}.\varphi.\sigma$ is satisfiable iff the $\bigwedge_{|\sigma|} \Gamma$ is satisfiable. Moreover, the size of $\bigwedge_{|\sigma|} \Gamma$ is linear in the size of σ.*

Example 8 [Constraints from Traces] Consider the infeasible trace from [BR02a] shown on the left in Figure 3.10. On the right, the figure shows the result of $\mathsf{Con}.(\theta_0, \Gamma_0 :: \mathit{true}).\sigma$, where the initial lvalue map θ_0 maps a, b, and c to 0. To the right of each command is the corresponding constraint. When we cut the trace at the fourth location, the resulting pair (φ^-, φ^+) consists of the conjunctions of the constraints from above and below the line, respectively. The interpolant in this case is $\langle a, 3\rangle \leq \langle c, 1\rangle - 2$, which upon cleaning yields the predicate $a \leq c - 2$. Notice that the constants common to both sides of the cut denote the values of the respective variables after the first four operations, and φ^- implies the interpolant. □

Programs with Pointers : PII

l	$\mathsf{SP}.\varphi.l$	$\mathsf{Con}.(\theta, \Gamma).l$
$(\mathtt{f}() : \cdot)$	φ	$(\theta, \Gamma :: \mathit{true})$
$(\mathtt{return} : \cdot)$	“”	“”
$(\mathtt{l} := \mathtt{e} : \cdot)$	$\exists M'.(\varphi[M'/M] \wedge M = \mathit{upd}(M', M'.l, M'.e))$ where M' is a fresh store	$(\theta', \Gamma :: \varphi)$ where $(\theta', \varphi) = \mathsf{Asgn}.X.\theta.(l, e)$ X is the program's variables
$(\mathtt{assume}[\mathtt{p}] : \cdot)$	$\varphi \wedge M.p$	$(\theta, \Gamma :: \mathsf{Sub}.\theta.(\mathsf{clos}^*.\mathit{true}.p))$

Figure 3.11: Postconditions and constraints for PII traces.

We now show how to generate constraints for consider programs with pointers in a sound and complete way.

Constraints for modeling allocation. Suppose there are two variables x and y, each of type `ref` *Int*. When the program begins, and the pointers are allocated, the standard semantics is that their values are not equal. For completeness, this must be explicitly modeled by constraints. For a set X of variables, let:

$$\mathsf{Mreach}.X = \{*^k x \mid x \in X \text{ and } k \geq 0\}$$

be the set of cells that are reachable from X by dereferences. As we do not have recursive types, this set is finite and syntactically computable (k is bounded by the type of x). For an lvalue l, let

$$\mathsf{Alias}.X.l = \{l' \mid l' \in \mathsf{Mreach}.X \text{ and } l' \text{ may be aliased to } l\}$$

Note that the above is a subset of $\{l' \mid l' \in \mathsf{Mreach}.X \text{ and } \mathsf{typ}.l' = \mathsf{typ}.l \neq \mathit{Int}\}$, and hence, finite. For correctness, we shall only require that $\mathsf{Alias}.X.l$ overapproximates the set of lvalues that can actually alias l. Let

$$\mathsf{Aliases}.X = \{(l, l') \mid l \in \mathsf{Mreach}.X \text{ and } l' \in \mathsf{Alias}.X.l\}$$

To model the distinctness of pointers at the start of execution, we assume that the first command of the initial function $f_{\mathtt{main}}$, is an assume statement that stipulates the predicate $\bigwedge_{(l,l') \in \mathsf{Aliases}.X} l \neq l'$, where X is the program's variables. Note that this clause is this clause is quadratic in the size of X. An example is the first `assume` in the trace of Figure 3.12.

Modeling the store with lvalue maps. Recall that using *sel* and *upd* it is straightforward to generate the strongest postconditions for programs with pointers; see Figure 3.11. Unfortunately, the theory of arrays does not have the interpolant property, thus we cannot get interpolants from TFs that use this theory. For example, the conjunction of:

$$M' = \mathit{upd}(M, x, y)$$

and

$$(a \neq b) \wedge (\mathit{sel}(M, a) \neq \mathit{sel}(M', a)) \wedge (\mathit{sel}(M, b) \neq \mathit{sel}(M', b))$$

is not satisfiable, but there is no quantifier-free interpolant in the common set of variables, namely $\{M, M'\}$. We surmount this hurdle by modeling the memory axioms using (generalized) lvalue maps, and by instantiating the array axioms on demand. Recall the definitions of lvalue maps and Upd from Section 3.3.3. The set ChLval consists of elements cl generated by the grammar $cl ::= \langle x, i\rangle \mid \langle cl, i\rangle$, where $i \in \mathbb{N}$. The function Clean of the previous section is extended by $\mathsf{Clean}.\langle x, i\rangle = x$ and $\mathsf{Clean}.\langle cl, i\rangle = *(\mathsf{Clean}.cl)$. Each $cl \in \mathit{ChLval}$ is a special constant that denotes the value of $\mathsf{Clean}.cl$ at some point in the trace. The function Sub of the previous section is extended to all lvalues by $\mathsf{Sub}.\theta.(*^k x) = \langle x, \theta.x\rangle$ if $k = 0$, and $\mathsf{Sub}.\theta.(*^k x) = \langle \mathsf{Sub}.\theta.*^{k-1} x, \theta.(*^k x)\rangle$ otherwise, and extended naturally to expressions, atomic predicates, and formulas.

Constraints for assume operations. Modeling the memory with *sel* and *upd* gives us some relations for free, *e.g.* from $x = y$ (modeled as $\mathit{sel}(M, x) = \mathit{sel}(M, y)$) the equality $*x = *y$ (modeled as $\mathit{sel}(M, \mathit{sel}(M, x)) = \mathit{sel}(M, \mathit{sel}(M, y))$) follows by congruence. We

explicitly state these implied equalities when generating constraints, by closing a predicate with the operator $\mathsf{clos}^*.true\colon FOL \to FOL$, where

$$\mathsf{clos}^*.b.p = \begin{cases} (\mathsf{clos}^*.b.p_1)\ \mathsf{op}\ (\mathsf{clos}^*.b.p_2) & \text{if } p \equiv (p_1\ \mathsf{op}\ p_2), \\ \neg(\mathsf{clos}^*.(\neg b).p_1) & \text{if } p \equiv (\neg p_1), \\ p \wedge \bigwedge_{0 \le k \le N}((*^k l_1) = (*^k l_2)) & \text{if } p \equiv (l_1 = l_2) \text{ and } b = true \\ p & \text{otherwise.} \end{cases}$$

provided $\mathsf{typ}.l_1 = \mathsf{typ}.l_2 = \mathtt{ref}^N Int$. The formula $\mathsf{clos}^*.true.p$ explicates all equalities inferred by the memory axioms from the formula p. When generating the constraints for `assume` (p), we first "close" p using clos^*, and then generate constraints for the result. Consider, for example, the constraint for the fourth command in Figure 3.12. For any formula p that can appear in an `assume` , we have $M.p \Leftrightarrow M.(\mathsf{clos}^*.true.p)$ in the theory of arrays. Using this equivalence, we can show the following proposition, which tells us that the constraints have been modeled adequately. For a program P, an lvalue $*^k x$ is well-typed in P if $\mathsf{typ}.x = \mathtt{ref}\ ^N Int$ for some $N \ge k$, *i.e.,* if x has type $\mathtt{ref}\ Int$, then $*x$ is well-typed but not $***x$. A formula p is well-typed w.r.t. P if (1) it does nor contain memory variables, *sel*, or *upd*, and (2) each lvalue that occurs in p is well-typed in P.

Proposition 5 *For a program P, two formulas $p, p' \in \mathsf{Pred}.X$ that are well-typed w.r.t. P, and an lvalue map θ, the condition $M.p$ implies $M.p'$ iff $\mathsf{Sub}.\theta.(\mathsf{clos}^*.true.p)$ implies $\mathsf{Sub}.\theta.p'$.*

Constraints for assignments. When assigning to $*l_1$ we must explicate that for all lvalues $*l_2$ such that $l_1 = l_2$, the value of $*l_2$ is updated as well. Let Equate be a function that takes a pair of lvalue maps (θ_1, θ_2) and a pair of expressions (l_1, l_2), and generates equalities between the names of l_1 and its transitive dereferences under θ_1, and the names of l_2 and its transitive dereferences under θ_2. Formally,

$$\mathsf{Equate}.(\theta_1, \theta_2).(l_1, l_2) = \bigwedge_{0 \le k \le N} (\mathsf{Sub}.\theta_1.(*^k l_1) = \mathsf{Sub}.\theta_2.(*^k l_2)),$$

where $\mathsf{typ}.l_1 = \mathsf{typ}.l_2 = \mathtt{ref}^N Int$. Define the function EqAddr, which takes a pair of lvalues and returns a formula that is true when the lvalues have the same address, as:

$$\mathsf{EqAddr}.(*^{k_1} x_1, *^{k_2} x_2) = \begin{cases} false & \text{if } k_1 = 0 \text{ or } k_2 = 0 \\ (*^{k_1 - 1} x_1 = *^{k_2 - 1} x_2) & \text{otherwise.} \end{cases}$$

Finally, we define the function Asgn, which generates appropriate constraints for an assignment `l := e`. The function Asgn takes a set of variables X, an lvalue map θ, and a pair (l, e), where l is an lvalue and e the expression that is being written into l, and returns a pair (θ', φ') of an updated lvalue map θ' and a formula φ'. Define $\theta' = \mathsf{Upd}.\theta.S$, where $S = \{*^k l' \mid l' \in (\mathsf{Alias}.X.l) \cup \{l\} \text{ and } k \geq 0\}$, and define

$$\varphi' = \mathsf{Equate}.(\theta', \theta).(l, e) \wedge \bigwedge_{l' \in \mathsf{Alias}.X.l} \left(\begin{array}{ll} \mathsf{ite}. & (\mathsf{Sub}.\theta.(\mathsf{EqAddr}.(l, l'))). \\ & \quad (\mathsf{Equate}.(\theta', \theta).(l', e)). \\ & \quad (\mathsf{Equate}.(\theta', \theta).(l', l')) \end{array} \right).$$

The first conjunct of φ' states that l gets a new value e, and all transitive dereferences of l and e are "equated" (*i.e.,* $*l$ gets the new value $*e$, and so on). The big second conjunct of φ' states how the potential aliases l' of l are updated: if l and l' have the same address, then the new value of l' (given by $\mathsf{Sub}.\theta'.l'$) is equated with e; otherwise the new value of l' is equated with the old value of l' (given by $\mathsf{Sub}.\theta.l'$). This generalizes Morris' definition for the strongest postcondition in the presence of pointers [Mor82].

Proposition 6 *Let $l := e$ be an assignment in a program P, let $\varphi = \mathsf{SP}.true.(\mathtt{l} := \mathtt{e} : \cdot)$, and let $(\theta', \varphi') = \mathsf{Con}.(\theta_0, true).t$ for some lvalue map θ_0. For every formula $p \in \mathsf{Pred}.X$ that is well typed w.r.t. P, the formula φ implies $M.p$ in the theory of arrays iff φ' implies $\mathsf{Sub}.\theta'.p$.*

Constraint Generation. Figure 3.11 gives the definition of the operator SP using the theory of arrays, as well as the generated constraints. Notice that the "current" memory is always represented by M. We use Asgn to generate the appropriate constraints for dealing with the possible alias scenarios. For assume operations, the constraint generated is on the "closure" of the predicate using clos^*. Constraints for traces are obtained as before. The initial predicate $\mathsf{IC}.\theta_0.\varphi$ is $\mathsf{Sub}.\theta_0.(\mathsf{clos}^*.true.\varphi)$. The size of the constraints is quadratic in the size of the trace. By induction over the length of the trace, splitting cases on the kind of the last operation, and using Propositions 5 and 6, we can prove the following theorem.

Theorem 5 *Given a trace σ of a program P, let $(\theta, \Gamma) = \mathsf{Con}.(\theta_0, \Gamma_0 :: \mathsf{IC}.\theta_0.\varphi).t$, and let $\varphi_r = \mathsf{SP}.\varphi.\sigma$. For every formula $p \in \mathsf{Pred}.X$ that is well-typed w.r.t. P, the formula*

$$
\begin{array}{rl}
 & \textit{true} \\
(\texttt{assume}\ \ (x \neq y) : pc_1); & \langle x, 0\rangle \neq \langle y, 0\rangle \\
(*x := 0 : pc_2); & \langle *\langle x, 0\rangle, 3\rangle = 0 \\
(y := x : pc_3); & \langle y, 4\rangle = \langle x, 0\rangle \wedge \langle *\langle y, 4\rangle, 5\rangle = \langle *\langle x, 0\rangle, 3\rangle \\
(\texttt{assume}\ (y = x) : pc_4); & \langle y, 4\rangle = \langle x, 0\rangle \wedge \langle *\langle y, 4\rangle, 5\rangle = \langle *\langle x, 0\rangle, 3\rangle \\
(*y := *y + 1 : pc_5); & \langle *\langle y, 4\rangle, 6\rangle = \langle *\langle y, 4\rangle, 5\rangle + 1 \wedge \\
 & \quad \mathsf{ite}.(\langle x, 0\rangle = \langle y, 4\rangle) \\
 & \quad\quad .(\langle *\langle x, 0\rangle, 7\rangle = \langle *\langle y, 4\rangle, 5\rangle + 1) \\
 & \quad\quad .(\langle *\langle x, 0\rangle, 7\rangle = (\langle *\langle x, 0\rangle, 3\rangle)) \\
(\texttt{assume}\ (*x = 0) : pc_6) & \langle *\langle x, 0\rangle, 7\rangle = 0
\end{array}
$$

Figure 3.12: Cutting a PII trace.

φ_r implies $M.p$ in the theory of arrays iff $\bigwedge_{|\sigma|} \Gamma$ implies $\mathsf{Sub}.\theta'.p$. Hence, the trace φ_r is satisfiable iff $\bigwedge_{|\sigma|} \Gamma$ is satisfiable. Moreover, the size of $\bigwedge_{|\sigma|} \Gamma$ is cubic in the size of σ.

Example 9 [Constraints from PII Traces] The right column in Figure 3.12 shows the constraints for the trace on the left. For readability, we omit unsatisfiable and uninteresting disjuncts (for the second and third commands). At the fourth cut-point of this trace, the common variables are $\langle *\langle y, 4\rangle, 3\rangle$, $\langle y, 4\rangle$, $\langle x, 0\rangle$, and $\langle *\langle x, 0\rangle, 3\rangle$, which denote the values of $*y$, y, x, and $*x$ at that point in the trace. The interpolant for this cut is $\langle *\langle y, 2\rangle, 3\rangle = 0$, which gives the predicate $*y = 0$ for the location pc_4. $\square$

3.4 Theoretical Issues

In this section we consider two theoretical issues regarding lazy abstraction. First, we provide sufficient conditions for the termination of the algorithm $\mathsf{LazyAbstraction}$. Second, we show that it is undecidable to check if there is a finite predicate abstraction that is sufficient to prove a given safety property.

3.4.1 Termination

Let $\mathcal{S} = (X, \Sigma, \rightsquigarrow)$ be a labeled transition system. For a state $s \in S$ and a sequence $\sigma \in \Sigma^*$, we write $s \overset{\sigma}{\rightsquigarrow}$ if there is a state $s' \in S$ such that $s \overset{\sigma}{\rightsquigarrow} s'$. Two states $s_1, s_2 \in S$ are *trace-equivalent* if for every $\sigma \in \Sigma^*$, we have $s_1 \overset{\sigma}{\rightsquigarrow}$ iff $s_2 \overset{\sigma}{\rightsquigarrow}$. The labeled transition system $\mathcal{S}$ has a *finite trace equivalence* if the trace-equivalence relation on S has a finite index.

Let $\mathcal{A}$ be an abstraction structure for $\mathcal{S}$ with region set R and extension function $[\![\cdot]\!]$. The abstraction structure $\mathcal{A}$ satisfies the *ascending-chain condition* if there does not exist

an infinite strictly increasing sequence $r_0 \sqsubset r_1 \sqsubset \cdots \sqsubset r_k \sqsubset \cdots$ of regions in R, where $r \sqsubset r'$ if $[\![r]\!] \subset [\![r']\!]$.

In the following theorem we make two assumptions. First, in order to relate trace equivalence with the reachability of error states, we assume without loss of generality that error states have no outgoing transitions; that is, for every state $s \in [\![\mathcal{E}]\!]$, there is no label $l \in \Sigma$ such that $s \xrightarrow{l}$. Second, we assume that the `keep_subtree` function used by algorithm LazyAbstraction on line 11 always returns *false*, to avoid infinite loops as in Example 2.

Theorem 6 [Termination] *Let $\mathcal{A}$ be an abstraction structure for a labeled transition system $\mathcal{S}$, and let Φ be a refine operator for $\mathcal{A}$. If*

(i) $\mathcal{S}$ has a finite trace equivalence, and

(ii) $\mathcal{A}$ satisfies the ascending chain condition,

then for every initial region r_0 and error region $\mathcal{E}$, the execution of LazyAbstraction.$(\mathcal{A}, \Phi, r_0, \mathcal{E})$ *(Algorithm 2) terminates.*

In the proof, we use finite trace equivalence to show that a node in the reachability tree cannot be refined infinitely often, and then derive (by way of contradiction) an infinite ascending chain of regions for any nonterminating run. Unfortunately, the regions obtained from predicate abstraction with respect to an infinite predicate language usually do not satisfy the ascending chain condition. However, for a given labeled transition system with a finite trace equivalence, we may be able to choose a predicate language with a *finite* set of predicates, such as predicates that define (unions of) trace-equivalence classes. For example, this is the case for timed automata [AD94]. As the boolean combinations of a finite set of predicates trivially satisfy the ascending chain condition, the theorem guarantees termination.

3.4.2 Finite predicate abstraction is undecidable

The Lazy Abstraction algorithm with predicate abstraction does not necessarily terminate on labeled transition systems with infinite state spaces. Indeed, we show that the problem whether there is a finite set of support predicates that witnesses a given safety property is

undecidable. Let $\mathcal{L}$ be a predicate language for the labeled transition system $\mathcal{S} = (X, \Sigma, \rightsquigarrow)$. Let Γ be a finite set of predicates from $\mathcal{L}$, and define the induced equivalence $\cong_\Gamma$ on S as $s_1 \cong_\Gamma s_2$ iff for all predicates $p \in \Gamma$, we have $s_1 \in [\![p]\!]$ iff $s_2 \in [\![p]\!]$; denote by $[s]_{\cong_\Gamma}$ the equivalence class of state s. The *quotient* $\mathcal{S}_{\cong_\Gamma}$ is the labeled transition system $(\{x_\cong\}, \Sigma, \rightsquigarrow_\cong)$, with the single variable $x_\cong$ whose range is the (finite) set of equivalence classes of $\cong_\Gamma$, and for all $l \in \Sigma$, we have $s \overset{l}{\rightsquigarrow}_\cong s'$ iff there exist two states $t \in s.x_\cong$ and $t' \in s'.x_\cong$ with $t \overset{l}{\rightsquigarrow} s'$. Note that every path in labeled transition system has a counterpart in the quotient, but not necessarily vice versa.

For a predicate language $\mathcal{L}$ for 2-counter machines, the *$\mathcal{L}$-finite abstraction problem* $\mathcal{L}$-FINABS is defined as follows:

- **Input** A 2-counter machine M, an initial state m_0 and a final state m_f of M, both definable in $\mathcal{L}$.

- **Output** "Yes" if either m_f is reachable in M from m_0, or there is a finite set Γ of predicates from $\mathcal{L}$ such that $[m_f]_{\cong_\Gamma}$ is not reachable in the quotient $M_{\cong_\Gamma}$ from $[m_0]_{\cong_\Gamma}$.

Notice that the problem is not trivial as the set of states reachable from m_0 (or the set of states that can reach m_f) may not be expressible as a boolean formula over predicates in $\mathcal{L}$.

Let Presburger-FINABS be the finite abstraction problem where $\mathcal{L}$ contains the control locations as propositions, and the quantifier-free formulas of Presburger arithmetic for constraining the counter values. We show undecidability by reduction from the halting problem for 2-counter machines. In particular, given M, we construct a 2-counter machine M' with initial state m'_0 and halting state m'_f such that M halts iff $\langle M', m'_0, m'_f \rangle$ is in Presburger-FINABS (we construct M' such that the reachable states of M' cannot be defined by a Presburger formula).

Theorem 7 *Presburger-FINABS is complete for Σ^0_1 sets.*

More generally, let $\mathcal{L}$ be any predicate language for 2-counter machines. Then $\mathcal{L}$-FINABS is complete for Σ^0_1 sets.

3.5 Related work

Our work is related to counterexample-driven abstraction refinement [BR01; CGJ+00; DD01; Sai00]. As in [BMMR01; DDP99; GS97a], we automatically construct a predicate abstraction by using an automatic theorem prover to answer satisfiability queries. However, all previous counterexample-driven refinement methods do not reuse the work done in one pass in the next pass: after every pass, the abstraction is constructed from scratch, and the new system is model checked. The results from model checking the previous passes are not reused, and a large part of the symbolic state space may be traversed repeatedly, even though a coarser abstraction is sufficient to prove the property of interest for that region. Lazy abstraction takes advantage of previous runs by abstracting locally.

Dataflow and type-based analyses have been used to check safety properties of systems code (e.g., [ECCH00; FTA01; Ste93]). These analyses typically ignore data dependence and may generate false positives owing to infeasible paths. Our work can be seen as an extension to such analyses by introducing path sensitivity to the analysis. Moreover, counterexample-driven refinement avoids an explosion of spurious error traces.

Current predicate discovery engines that learn predicates from traces implement (variations on) the following basic algorithm [HJMS02; BR02a; DD02; CCGS03]. Given a trace, the tool symbolically executes the trace, keeping a history of what values every variable held at each point, and checking for an inconsistency at each step. Upon finding an inconsistency, the tool looks at which components of the present state are inconsistent, and performs a value flow analysis to detect which old values can transitively effect the inconsistent state. It then returns a minimally infeasible subset of the facts as the new set of predicates. This method suffers from two drawbacks. First, the inconsistent state may depend on old values of variables, requiring the introduction of new "symbolic variables" denoting these stale values for the variables at various points in the program []. However it is often not clear which symbolic variables are enough, and several ad hoc heuristics are used [BR]. Second, the tool cannot pinpoint where exactly the predicates are needed and where they are not, thus making the resulting abstraction over several iterations too detailed; efforts to minimize the set over several iterations lead to considerable optimization effort [CCGS03].

On the other hand, predicate discovery based on interpolants does not have the above problems: the predicates learnt are in terms of the current state (thus eliminating the need for intermediate symbolic variables in functions), and *local* (thus keeping the abstraction small).

Chapter 4

Applications

We have implemented Lazy Abstraction in a tool called BLAST , downloadable at http://www.eecs.berkel
that checks safety properties of C programs. We handle all syntactic constructs of the C language, including pointers, structures, and procedures. Constructs not in the predicate language are left uninterpreted, which implies a loss of precision in our "concrete" semantics. Also, we do not model pointer arithmetic precisely; we assume a *logical* model of memory. Thus, we model the expression $p + i$, where p is a pointer and i is an integer, as yielding a pointer value that points to the object pointed to by p. Currently we handle procedure calls using an explicit stack and do not handle recursive functions, but the systems code we have analyzed is not recursive.

Our tool is written in Objective Caml (http://www.ocaml.org), and consists of two main parts:

1. A functor implementing the LazyAbstraction algorithm, which takes a symbolic abstraction structure together with a focus operator as input, and

2. The symbolic abstraction structure and refine operator for C. The latter is made up of two parts: (a) the C front end, for which we use the CIL C Compiler Infrastructure [NMRW02], which converts a C program to a Control-flow Graph, from which we obtain the CFAs, and (b) a module that contains the data structures for C regions as well as the functions post, $\widehat{\mathsf{post}}$, and Refine.

In order to obtain checkers for other formalisms, one need only to implement and plug in a symbolic abstraction structure together with a Refine procedure for the formalism. We briefly discuss some aspects of our implementation, not considered in earlier chapters.

1. Regions. The boolean formulas over predicates that represent data regions are stored as BDDs [Som98] to get a canonical sum-of-product form for formulas. Apart from compactly representing the abstract regions, the BDD representation also allows easy boolean manipulation and covered (inclusion) checking. For each control state, *i.e.,* pair of program counter and stack, we store as a BDD, the *union* of all the data regions seen for that control location. To check if a node is covered, we use BDD operations to check whether the node's data region implies (under a purely boolean interpretation), the presently seen data region for the corresponding control location. To check if a region is empty, we check, using the decision procedure Simplify [DNS], whether the formula corresponding to the region is satisfiable.

2. Post. To implement SP, we use a flow-insensitive implementation of Andersen's Alias analysis [And94] to prune the number of disjuncts arising from splitting cases on the possible lvals that may be affected by an update (cf. Section 3.3.3).

3. Abstract Post : Cartesian Abstraction. We have found that computing the most precise predicate abstraction via a recursive subdivision of the state space [GS97b; DDP99; SS99; FQ02] is prohibitively expensive for the large numbers of predicates that we need to track, as they require exponentially many (in the number of predicates), calls to decision procedures. Fortunately, for the programs we have considered, a significantly cheaper method, called *cartesian abstraction*[GS97b] is sufficiently precise. We fix a canonical sum-of-product form for formulas. We ask, for each disjunct ψ of the canonical sum-of-product form of φ, and each support predicate $p \in \Lambda$, if ψ implies p, and if ψ implies $\neg p$. This gives, for each disjunct ψ of φ, a conjunction ψ' of support predicates, where the predicate p occurs positively if $\psi \Rightarrow p$ is valid, occurs negatively if $\psi \Rightarrow \neg p$ is valid, and does not occur if neither validity holds. Let φ' denote the disjunction of all conjunctions ψ' so constructed. The region $\widehat{\mathsf{Abs}}.\Lambda.\varphi$ is the formula φ'. Hence, to compute $\widehat{\mathsf{post}}$, we compute

$$\mathsf{SP}_{\Pi}.\varphi.l = \widehat{\mathsf{Abs}}.(\Pi.l).(\mathsf{SP}.\varphi.l)$$

The imprecision of Cartesian abstraction has not caused false positives for the programs we have considered. In our experiments (see Section 4.1), we could prove all the properties using

the Cartesian $\widehat{\mathsf{post}}$ in much less running time. Since the abstract $\widehat{\mathsf{post}}$ of a region is computed very frequently in the lazy-abstraction algorithm, any speedup in its computation results in a significant overall speedup. The Cartesian abstraction computed above takes time linear in the number of support predicates, as opposed to exponential in the number of support predicates for the most precise computation. For the validity checks described above, checks in the counterexample analysis we use the theorem prover Simplify. Nevertheless, it would be interesting to see if the interpolation based methods described in Section 3.3 could be applied to refine Cartesian Abstractions by discovering which predicates need to be tracked precisely. In addition to the above, we treat *basic blocks* of sequences of assignments.

4. Refine. The algorithm for generating interpolants [McM04] uses the VAMPYRE proof-generating theorem prover, available at `http://www.eecs.berkeley.edu/~rupak/Vampyre`. For efficiency, we have implemented several optimizations of the basic procedure described in Section 3.3. First, we treat sequences of assignments atomically. Second, we do not cut at every point of a spurious error trace. Instead, we perform a preliminary analysis which identifies a subset of the constraints that imply the infeasibility of the trace, and only consider the instructions that generate these constraints as cut-points. It is easy to check that the optimized procedure is still complete. We also identify multiple reasons for infeasibility of a trace in one refinement step. When we have identified a subset of constraints that imply infeasibility, we replace each constraint arising from an assume statement in this subset to *true*, and repeat a search for unsatisfiability with the new, weaker set of constraints.

4.1 Device Driver Verification

We have run the implementation to check simple safety properties of some Linux and Microsoft Windows NT device drivers. The properties are specified using a language that allows the user to specify legal sequences of events (such as function calls, returns), using monitor automata that can view those events during the execution of the program. The safety property then amounts to specifying which sequences of events are legal: illegal sequences of events lead to the error state. The language is described in detail in [BCH+04a]. By default BLAST makes optimistic assumptions about missing functions, namely they are

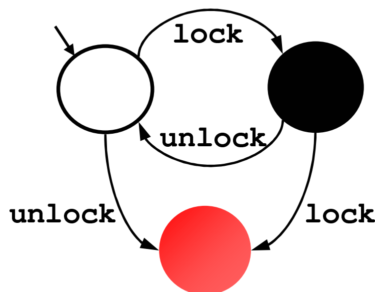

Figure 4.1: Property 1: Double Locking

.

Name	LOC	Predicates		Thm Prover Calls		Running
		Total	Active	Total	Cached	Time (s)
`driver.c`	95	3	3	260	165	0.08
`funlock.c`	40	4	3	340	182	0.14
`read.c`	370	28	18	5643	2862	4.42
`floppy.c`	6473	5	5	4137	3759	2.05
`qpmouse.c`	400	3	3	3117	2925	0.74
`ll_rw_block.c`	1281	9	7	10143	9483	5.82

Table 4.1: Double Locking

treated as `assume` *true*. We check the code using a model of the kernel that exercises the driver. The model first calls the driver initialization routine, then calls the driver functions (read, write, etc.) in a loop, and finally unloads the driver.

Double Locking

The first property we checked for was double locking, *i.e.,* that calls to `lock()` and `unlock()` alternate, the same property as in the example from Section 3.1, shown again in Figure 4.1. All times are on a 800MHz Pentium III with 256M RAM, and do not include parsing time. The results are summarized in Table 4.1. LOC refers to *postprocessed* lines of code. The total number of predicates is the total number required in the run; the active column gives the total number of support predicates active at any particular node in the reachability tree. There are many redundant theorem prover calls (the fraction of cached calls is very high). In several examples, the benefits of local predicates can be seen as the number of active predicates is less than the total number of predicates. This is especially true for `read.c`, because the property being checked has two disjoint branches, which require different sets of predicates to be verified.

The program **driver.c** is the driver code from [BR01]; the program **funlock.c** is the example from Section 3.1. The file **read.c** is a (simplified) serial driver and **floppy.c** is a floppy driver from the Microsoft Windows DDK. Finally, **qpmouse.c** and **ll_rw_block.c** are Linux device drivers (from the 2.4.9 kernel). We check locking disciplines in **floppy.c** and **ll_rw_block.c**. We check for null pointer dereferences in **qpmouse.c**. In **read.c** we check the property discussed in [BR01], namely, that the driver dispatch routine correctly handles both immediate and asynchronous services. In most cases, correctness cannot be proven using a data-independent analysis [FTA01], and requires the automatic discovery of relevant predicates. Moreover, correctness spans several functions, so an interprocedural analysis is required.

In **ll_rw_block.c**, a spinlock is acquired in function **__make_request**, and passed on to function **add_request**. Under normal circumstances, **add_request** returns with the lock held, and **__make_request** unlocks it. However, in case there is a system bug, the driver invokes the macro `BUG()`, and the lock is unlocked. The first run of the tool did not model the behavior of `BUG()` (which causes the system to crash), and found an error trace that involved a call to **add_request** from **__make_request** with the lock held, a system bug, an unlock and return, and a subsequent unlock in **__make_request**. We then modified the specification to check for the locking discipline only when no system bugs occur. The property could now be proved.

Program	LOC	Monolithic		Parsimonious			
	src/pre	Disc	Reach	Disc	Reach	Preds	Avg/Max
kbfiltr	5933/12301	1.12	0.30	3.48	0.10	72	6.5/16
floppy	8570/17707	7.10	3.59	25.20	0.46	240	7.7/37
diskperf	7209/14286	5.36	3.3	13.32	0.27	140	10/31
cdaudio	8921/18209	20.18	4.55	23.51	0.52	256	7.8/27
parport	12288/61777	-	-	74.58	2.23	753	8.1/32
parclass	30380/138373	-	-	77.40	1.6	382	7.2/28

Table 4.2: Experimental results for Blast checking the IRP Handling property: 'm' stands for minutes, 's' for seconds; 'LOC' is the number of lines of code: src is raw C source, pre is preprocessed code fed into Blast (including stubs, libraries etc.) 'Monolithic' is using the same set of predicates everywhere, previous approaches to predicate discovery, 'Parsimonious' is the method described in the previous chapter; 'Disc' is the total running time of the verification starting with the empty set of predicates; 'Reach' is the time to perform the reachability analysis only, given all necessary predicates; 'Preds' is the total number of predicates required, and Avg (Max) is the average (maximum) number of predicates tracked at a program location; the symbol '-' indicates that the tool does not finish in 6 hours.

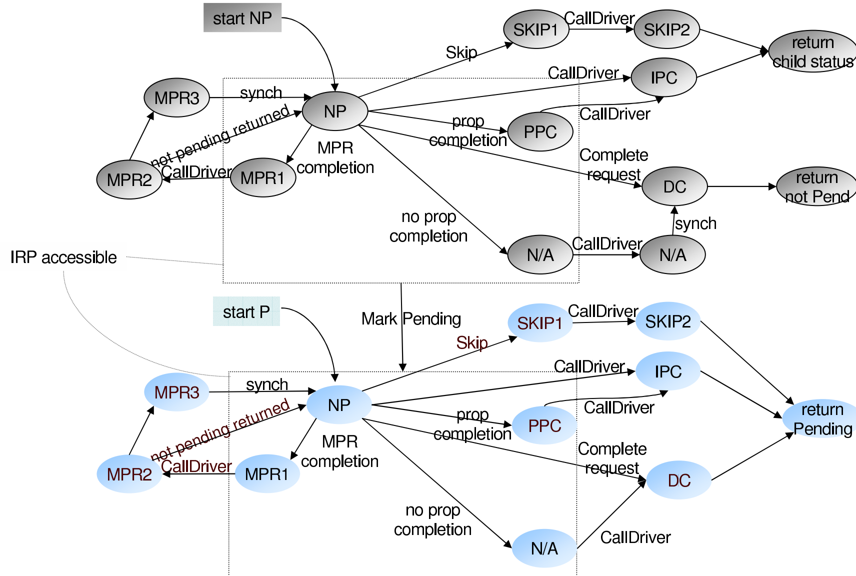

Figure 4.2: Property 2: IRP Handler

IRP Handler

The second safety property we considered was related to the correct handling of I/O Request Packets (IRP) by Windows NT device drivers. The property is a finite-state automaton with 22 states [BR], shown in Figure 4.2. The states correspond to various stages in the processing of the IRP, and the edges to kernel system calls. The results, obtained on an IBM ThinkPad T30 laptop with a 2.4 GHz Pentium processor and 512MB RAM, are summarized in Table 4.2. We present two sets of numbers: 'Monolithic' gives the times for using a monolithic set of predicates (traditional predicate abstraction) throughout the state space; 'Parsimonious' uses interpolants and tracks only the relevant predicates at each program location. The monolithic version of Blast timed out after several hours on the drivers `parport` and `parclass`. We found several violations of the specification in `parclass`. The numbers in the table refer to a version of `parclass` where the cases that contain errors are commented out. 'Parsimonious' performs better than the monolithic version (for larger programs), as here the cost of predicate/location discovery is more than made up for by the saving via the reduced abstract state space. For smaller programs, when started with the empty set of initial predicates, 'Monolithic' is faster than 'Parsimonious', because the latter may rediscover the same predicate at several different program locations. However, even

for small programs, as the predicates are tracked extremely precisely (the average number of predicates at a program location is much smaller than the total number of predicates required), 'Parsimonious' uses considerably less memory, and subsequent runs (for example, for verifying a modified version of the program [HJMS03], or to generate proofs (described in the next section), and the proof trees much smaller.

4.2 Temporal-safety proofs from Reachability Trees

Proof-carrying code (PCC) [Nec97b] has been proposed as a mechanism for witnessing the correct behavior of untrusted code. Here, the code producer sends to the consumer the code annotated with loop invariants and function pre- and postconditions, as well as a proof of correctness of a verification condition, whose validity guarantees the correctness of the code with respect to the specification. From the code and the annotations, the consumer can build the verification condition and check the supplied proof for correctness. The checking of the proof is much simpler than its construction. In particular, by encoding the proof, proof checking becomes a type-checking problem. Proof-carrying code has the advantages of avoiding trusted third parties, and of being tamper-proof, because tampering with either the proof or the code will result in an invalid proof. The main problem faced by PCC is that a user may have to supply annotations such as loop invariants. In [Nec97b] it is shown how loop invariants can be inferred automatically for proofs of type and memory safety, but the problem of inferring invariants for behavioral properties, such as temporal safety, remains largely open [Ern00].

We show that Lazy Abstraction can be used naturally and efficiently to construct small correctness proofs for temporal-safety properties in a PCC based framework. The proof generation is intertwined with the model-checking process: the data structures produced by lazy abstraction automatically supply the annotations required for proof construction, and provide a decomposition of the proof which leads to a small correctness certificate. In particular, using abstraction predicates only where necessary keeps the proof small, and using the model checker to guide the proof generation eliminates the need for backtracking, e.g., in the proof of disjunctions. Our strategy to generate proofs from model-checking runs is different from [Nam01; PZ01]. We exploit the structure of sequential code so that

the proof is an invariant for every control location, along with local checks for every edge of the control-flow graph that the invariants are sound. Both [Nam01; PZ01] work at the transition-system level. On the other hand, they generate proofs for properties more general than safety.

We have implemented proof generation in BLAST , and have used it to automatically construct proofs that various Linux and Windows device drivers satisfy certain temporal safety properties. The proofs are fairly small, and we believe this provides evidence that the approach can be used to construct small correctness certificates for low level systems code.

4.2.1 Overview

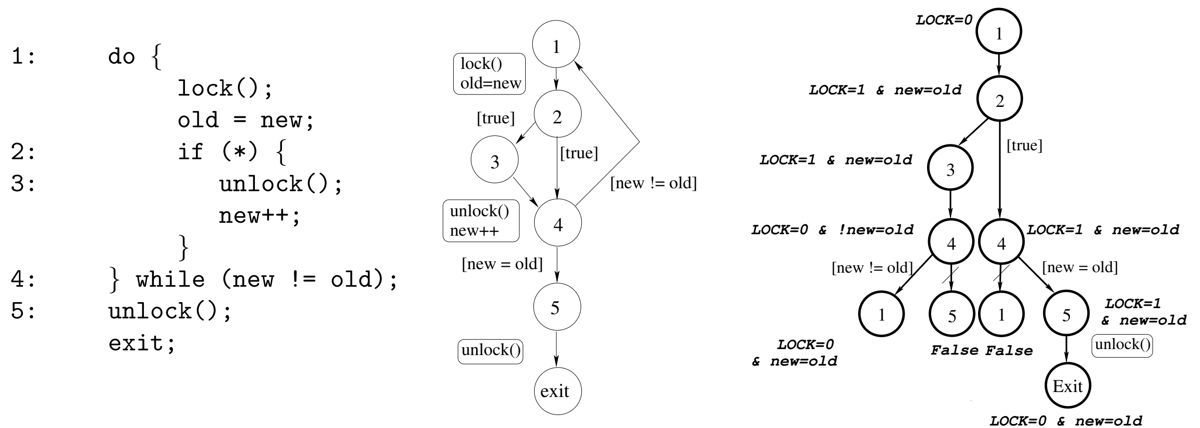

Figure 4.3: (a) The program `Example`, (b) CFA, (c) Reachability Tree

We consider a small example to give an overview of proof generation. We shall give a quick overview of proof generation using the program in Figure 4.3(a), which is a fragment of the program we saw earlier in Figure 2.2. The temporal-safety specification is that calls to `lock()`, and `unlock()` should alternate; as we saw before this specification is violated iff the program goes to the location `ERR`. Shown on the right in Figure 4.3(c) is the Reachability Tree produced by LazyAbstraction Algorithm 2 when invoked on this program with the error states `ERR`.

To certify that a program satisfies its specification, we use a standard *temporal-safety*

rule from deductive verification: given a transition system, if we can find a set I of states such that (1) I contains all initial states, (2) I contains no error states, and (3) I is closed under successor states, then the system cannot reach an error state from an initial state. If (1)–(3) are satisfied, then I is called an *inductive invariant* set. In our setting, the temporal-safety rule reduces to supplying for each vertex pc of the CFA an invariant formula $I.pc$ such that

1. $(LOCK = 0) \Rightarrow I.pc_0$;
2. $I.pc_{\mathcal{E}} = \mathit{false}$;
3. For each pair of CFA vertices $pc \xrightarrow{\mathsf{op}} pc'$ $\mathsf{SP}.(I.pc).(\mathtt{op} : pc') \Rightarrow I.pc'$.

Thus, to provide a proof of correctness, it suffices to supply a location invariant $I.pc$ for each vertex pc of the CFA, and proofs that the supplied formulas meet the above three requirements.

The location invariants can be mined from the reachability tree (Figure 4.3(c)). In particular, the invariant for pc is the disjunction of all reachable regions that label the nodes in the tree where the program counter is pc. For our example,

$$I.pc_4 = (LOCK = 0 \land \neg new = old) \lor (LOCK = 1 \land new = old)$$

It is easy to check, $(LOCK = 0) \Rightarrow I.pc_0$, since the root of the reachability tree is labeled by the precondition of the program $(LOCK = 0)$. Also, as there is no node labeled `ERR` in the tree, we get the second requirement by definition. The interesting part is checking that the third requirement, that for each edge $pc \xrightarrow{\mathsf{op}} pc'$ of the CFA, $\mathsf{SP}.(I.pc).(\mathtt{op} : pc') \Rightarrow I.pc'$. Consider the edge $pc_4 \xrightarrow{\mathtt{[new!=old]}} pc_1$. We need to show that

$$\mathsf{SP}((LOCK = 0 \land \neg new = old) \lor (LOCK = 1 \land new = old), [\mathtt{new!} = \mathtt{old}]) \Rightarrow (LOCK = 0).$$

To prove this, notice that as SP distributes over $\lor$, the disjuncts on the LHS can be broken down into subformulas obtained from individual tree nodes. We can show the implication by matching up each subformula with the appropriate successor on the RHS.

That is, it suffices to show that:

$$\mathsf{SP}((LOCK = 0 \wedge \neg new = old), [\texttt{new!} = \texttt{old}]) \Rightarrow (LOCK = 0)$$

$$\mathsf{SP}((LOCK = 1 \wedge new = old), [\texttt{new!} = \texttt{old}]) \Rightarrow (LOCK = 0)$$

To get each of the above, we recall how the formulas for the nodes of 1 were constructed when building the reachability tree — they were the predicate abstractions, *i.e.,* overapproximations computed using a theorem prover, of the strongest postconditions of the formulas labeling the respective parents. The computations by the theorem prover can be easily turned into proofs, which serve as proofs for the above obligations. We formalize this mechanism in the subsequent sections, and show how to generate machine-checkable proofs for correctness.

4.2.2 Verification Conditions

For ease of exposition, we describe our method only for programs (and CFAs) without function calls; it can be extended to handle function calls in a standard way (and function calls are handled by the BLAST implementation).

Let $C = (X, PC, pc_0, \rightarrow)$, φ_0 be a formula in $\mathsf{Pred}.X$ and $pc_{\mathcal{E}}$ be a special *error location* in PC. The initial region r_0 is $(pc_0, (\varphi_0, \cdot), \epsilon)$, and the error region $\mathcal{E}$ is $(pc_{\mathcal{E}}, (true, \emptyset), \cdot)$. The corresponding safety verification problem is to check whether $\mathsf{Reach}.(\mathcal{S}_C).r_0 \sqcap \mathcal{E}$ is not empty, in other words, to check whether there is a feasible path in C, to $pc_{\mathcal{E}}$ from pc_0, with the initial variables satisfying φ_0. Note that every temporal-safety property can be reduced to the above.

A *verification condition* (VC) [Dij76] for a program and a specification is a first-order formula Ψ such that the validity of Ψ ensures that the program adheres to the specification. In order to produce the VC we require an invariant, a map $I : PC \rightarrow FOL$. We call $I.pc$ the *invariant formula* of pc. Given and invariants I: the verification condition is defined as: $\mathsf{VC}.(C, \varphi_0, pc_{\mathcal{E}}, I)$ that asserts the correctness of the CFA is defined

$$\mathsf{VC}.(C, \varphi_0, pc_{\mathcal{E}}, I) = (\varphi_0 \Rightarrow I.pc_0 \wedge (I.pc_{\mathcal{E}} = false)\ ; \wedge \bigwedge_{pc \xrightarrow{\mathsf{op}} pc'} \left(\mathsf{SP}.(I.pc).(\mathsf{op} : pc') \Rightarrow I.pc'\right)$$

Which contains one conjunct for each edge of C. Intuitively, the invariant formula of a location is an overapproximation of the states that system can be in while at that control

location. The union of the invariant formulas is an overapproximation of the reachable region of the CFA. The check for each edge ensures that the overapproximation is a fixpoint. The specification that the error control location $pc_{\mathcal{E}}$ is not reachable, is captured by requiring $I.pc_{\mathcal{E}} = \mathit{false}$. In other words, the VC states that the invariant of each location is an inductive overapproximation of the states that can be reached at that location, and that the error states are not reached. The following theorem states that a proof of the above VC suffices to guarantee the safety of the given program: (recall that for a *FOL* formula Ψ we write $\vdash \Psi$ to indicate the validity of Ψ).

Theorem 8 [Adequacy of VC based proofs] *For a CFA C, formula φ_0, an error control location $pc_{\mathcal{E}} \in C.PC$, and an invariant annotation $I : C.PC \rightarrow FOL$, if $\vdash \mathsf{VC}.(C, \varphi_0, pc_{\mathcal{E}}, I)$ then $\mathsf{Reach}.\mathcal{S}_C.[\![r_0]\!] \cap [\![\mathcal{E}]\!] = \emptyset$ where $r_0 = (C.pc_0, (\varphi_0, \emptyset), \epsilon)$ and $\mathcal{E} = (pc_{\mathcal{E}}, (\mathit{true}, \emptyset), \epsilon)$.*

In *Proof-Carrying Code (PCC)* [Nec97b], the code producer sends to the consumer the code as well as annotations in the form of loop invariants and function pre and postconditions. From the code and the annotations, the code consumer builds the verification condition, whose validity guarantees the correctness of the code with respect to the safety specification. The code producer is required to supply to the consumer a *proof* of the validity of the VC, which the consumer is required merely to check. The checking of the proof should be much simpler than the production; in particular by an encoding of the proof, proof-checking becomes a fast type-checking problem. The annotations we require are more than those required by PCC, which are just loop invariants. We can justify these as they are being produced automatically and will also help in making the proof of the VC smaller. In the next section we show how lazy abstraction can be used to produce the annotations and prove the resulting VC. Unlike in [Nec97b] where it is assumed that the code is given in binary format we shall assume that both the producer and the consumer are working at the level of the CFA.

4.2.3 VCs and Proofs via Lazy Abstraction

We now show how using Lazy Abstraction, we can get both the Invariants needed to generate the VC, and then, a proof of the validity of the resulting VC.

Invariants grow on Trees

The invariants required for the VC are mined from the reachability tree produced by LazyAbstraction, Algorithm 2. We assume that both the code producer and consumer are working at the CFA level.

As there are no control stacks in the present setting, we can those, as well as the predicate sets from the node markings, and assume that the nodes are marked $\mathtt{n}:(pc, \varphi)$.

Recall, that for the safety verification problem mentioned above, the lazy abstraction algorithm, Algorithm 2, upon termination, either produces a counterexample demonstrating the system is unsafe, or, produces a reachability tree for C that is safe w.r.t $(pc_{\mathcal{E}}, \mathit{true})$ from (pc_0, φ_0) (Theorem 3). In the latter case, recall that the tree has the following properties:

1. For every edge $\mathtt{n} : (pc, \varphi) \xrightarrow{(\mathsf{op}:pc')} \mathtt{n}' : (pc', \varphi')$, we have $\mathsf{SP}.\varphi.(\mathsf{op} : \cdot) \Rightarrow \varphi'$, as φ' is $\mathsf{Abs}.\Lambda.\mathsf{SP}.\varphi.(\mathsf{op}:\cdot)$, for some set of predicates Λ,

2. For every leaf node $\mathtt{n}:(pc, \varphi)$, there are internal nodes $\mathtt{n}_1:(pc, \varphi_1), \ldots, node_k:(pc, \varphi_k)$, such that $\varphi \Rightarrow \bigvee_{1 \leq i \leq k} \varphi_i$,

3. The root node is $\mathtt{n}_0:(pc_0, \varphi_0)$,

4. For every $\mathtt{n}:(pc_{\mathcal{E}}, \varphi)$ in the tree we have $\varphi \wedge \varphi_{\mathcal{E}}$ is unsatisfiable.

The first two conditions follow as the tree is complete, the latter two as it is safe.

The reachable regions that label the nodes of a safe reachability tree provide the invariant map; the invariant formula $I.pc$ the control location $pc \in C.pc$ is defined to be the union of all reachable regions of *internal* nodes of the reachability tree, labelled by pc.

$$I.pc = \bigvee_{\mathtt{n}:(pc,\varphi) \in \mathit{Intl}} \varphi$$

Proof Generation

We now show how to generate a proof of the validity of the verification condition $\mathsf{VC}.(C, \varphi_0, pc_err, I)$ resulting from the invariant map I described above.

Representing proofs. We encode the proof of the verification condition in LF [HHP93], so that proof checking reduces to a linear-time type-checking problem. The logic we encode in

And introduction $\dfrac{\vdash \varphi_1 \qquad \vdash \varphi_2}{\vdash \varphi_1 \wedge \varphi_2}$ andi Or introduction $\dfrac{\vdash \varphi_1}{\vdash \varphi_1 \vee \varphi_2}$ orir $\dfrac{\vdash \varphi_2}{\vdash \varphi_1 \vee \varphi_2}$ oril

Rules for implication $\dfrac{\vdash \varphi \Rightarrow \psi_1}{\vdash \varphi \Rightarrow \psi_1 \vee \psi_2}$ imp-mono-l $\dfrac{\vdash \varphi_1 \Rightarrow \psi \qquad \vdash \varphi_2 \Rightarrow \psi}{\vdash \varphi_1 \vee \varphi_2 \Rightarrow \psi}$ imp-dist

$\dfrac{\vdash \varphi \Rightarrow \varphi_1 \qquad \vdash \varphi \Rightarrow \varphi_2}{\vdash \varphi \Rightarrow \varphi_1 \wedge \varphi_2}$ imp-andi $\dfrac{\vdash \varphi_1 \Rightarrow \varphi_2 \qquad \vdash \varphi_2 \Rightarrow \varphi_3}{\vdash \varphi_1 \Rightarrow \varphi_3}$ imp-trans

Figure 4.4: First order proof rules used in constructing the proof

LF is first-order logic with equality and special relation and function symbols for arithmetic and memory operations. The encoding is standard [HHP93; Nec97b], and is omitted. The inference rules of the proof system include the standard introduction and elimination rules for the boolean connectives used in natural deduction with hypothetical judgments [Pfe97], together with special rules for equality, arithmetic, and memory operations. In Blast , proofs are represented in binary form using Implicit LF [NL98]. We use the proof encoding and checking mechanism of an existing PCC implementation to convert proofs from a textual representation to binary, and to check proofs.

Generating proofs. Given a reachability tree for CFA C that is safe w.r.t. $\mathcal{E}$ from φ_0, let I denote the resulting invariant map, described above. We must prove the three conjuncts of the corresponding verification condition $\mathsf{VC}.(C, \varphi_0, pc_{\mathcal{E}}, I)$, namely, that (1) the precondition implies the invariant of the initial location, (2) the invariant of the error location is false, and (3) the invariants are closed under postconditions. We prove each conjunct separately.

The first conjunct of the VC is $\varphi_0 \Rightarrow I.pc_0$. Since the root $\mathtt{n}_0 : (pc_0, \varphi_0$, we know that φ_0 is a disjunct of $I.pc_0$. Hence, the first conjunct of the VC follows from simple propositional reasoning. The second conjunct of the VC is $I.pc_{\mathcal{E}} = \mathit{false}$; this holds as the tree was safe w.r.t. $pc_{\mathcal{E}}$.

For the third conjunct, it suffices to show a proof obligation for each edge of the CFA. We use distributivity of postconditions and implication over disjunction to break the obligation for a CFA edge into individual obligations for the edges of the safe reachability tree that correspond to the CFA edge. Then we discharge the smaller proof obligations by relating them to the construction of the reachable regions during the search phase of the lazy-

abstraction algorithm.

Consider the edge $pc \xrightarrow{\mathsf{op}} pc'$ of C, and the corresponding proof obligation $\mathsf{SP}.(I.pc)(\mathsf{op}: pc') \Rightarrow I.pc'$. Recall that $I.pc$ is the union of all reachable regions of nodes of the tree labeled by pc. Since SP distributes over disjunction, it suffices to prove:

$$\left(\bigvee_{\mathtt{n}:(pc,\varphi)\in Intl} \mathsf{SP}.\varphi.(\mathsf{op}: pc') \right) \Rightarrow \left(\bigvee_{\mathtt{n}':(pc',\varphi')\in Intl} \varphi' \right)$$

,or equivalently, to prove:

$$\bigwedge_{\mathtt{n}:(pc,\varphi)\in Intl} \left(\mathsf{SP}.\varphi.(\mathsf{op}: pc') \Rightarrow \bigvee_{\mathtt{n}':(pc',\varphi')\in Intl} \varphi' \right)$$

Hence, it suffices to prove one obligation for each internal node labeled by pc. Each such internal node $\mathtt{n}:(pc,\varphi)$ has a unique $(\mathsf{op}: pc')$-son $\mathtt{n}':(pc',\varphi')$. This observation is essential for guiding the proof generation. We break the proof of $\mathsf{SP}.\varphi.(\mathsf{op}: pc') \Rightarrow I.pc'$ into two cases, corresponding to whether the unique $(\mathsf{op}: pc')$-son, $\mathtt{n}'$ is an internal node or a leaf node.

If $\mathtt{n}'$ is an internal node, then it suffices to prove $\mathsf{SP}.\varphi.(\mathsf{op}: pc') \Rightarrow \varphi'$. We generate a proof for this by considering the computation that put the edge from $\mathtt{n}$ to $\mathtt{n}'$ into the safe reachability tree. Assume that $\varphi = \bigvee R_i$, where each disjunct R_i is a cube, a conjunction of literals, where each literal is either an abstraction predicate or its negation. Then $\varphi' = \bigvee R_i'$, where for each i, the disjunct R_i' is computed as an overapproximate (abstract) successor region of R_i as follows: the literal p (resp., $\neg p$) appears in R_i' iff $\mathsf{SP}.R_i.(\mathsf{op}:\cdot) \Rightarrow p$ (resp., $\mathsf{SP}.R_i.(\mathsf{op}:\cdot) \Rightarrow \neg p$) is valid. We extract a proof of validity from the decision procedure used to check the implication when building the reach tree. We combine these proofs of validity with `imp-andi` to get a proof for $\mathsf{SP}.R_i.(\mathsf{op}:\cdot) \Rightarrow R_i'$, for each i, which using `imp-mono-l`, yields a proof for $\mathsf{SP}.R_i.(\mathsf{op}:\cdot) \Rightarrow \varphi'$. As SP distributes over disjunction, we use the rule `imp-dist` to combine the above proofs into a proof for $\mathsf{SP}.\varphi.(\mathsf{op}:\cdot) \Rightarrow \varphi'$.

If $\mathtt{n}'$ is a leaf, then we break the proof into three parts. First, we generate a proof for $\mathsf{SP}.\varphi.(\mathsf{op}:\cdot) \Rightarrow \varphi'$ as above. Second, we check why the node $\mathtt{n}'$ is a leaf of the safe reachability tree. There must be a set $S = \{\mathtt{n}_1' : (pc' : \varphi_1'), \ldots \mathtt{n}_k' : (pc' : \varphi_k')\}$ of internal nodes that covered $\mathtt{n}'$, *i.e.,* such that $\varphi' \Rightarrow \bigvee_{1\leq i\leq k} \varphi_i'$; this set can be obtained from the lazy-abstraction algorithm. We extract the proof of the above implication. Third, we notice

that $\bigvee_{1\leq i\leq k}\varphi'_i \Rightarrow I.pc'$, by the definition of $I.pc'$. These three proofs are combined using transitivity of implication (`imp-trans`) into a proof of $\mathsf{SP}.\varphi.(\mathtt{op}\!:\!pc') \Rightarrow I.pc'$.

Lazy Abstraction optimizes proof-generation in two ways. First, we use the intermediate steps of the model checker to break a proof that invariants are closed under postconditions into simpler proofs about the disjuncts that make up the invariants. Moreover, these proofs are available from the forward-search phase of the model checker. Second, we reduce the size of the proof by using a coarse, nonuniform abstraction sufficient for proving correctness, as provided by the lazy-abstraction algorithm. This eliminates predicates that are not essential to correctness and submits fewer and smaller obligations to the proof generator than would a VC obtained by direct symbolic simulation of all paths.

Example 10 We illustrate the construction of the proof on the example from Section 3.1. In particular, we show the proof for the edge $pc_4 \xrightarrow{[\mathtt{new!=old}]} pc_1$. The proof obligation for this edge is:

$$\mathsf{SP}.((LOCK = 0 \wedge \neg new = old) \vee (LOCK = 1 \wedge new = old)).[\mathtt{new!} = \mathtt{old}] \Rightarrow (LOCK = 0)$$

Equivalently, after taking the strongest postcondition, and distributing over $\vee$, we get $(LOCK = 0) \vee \mathit{false} \Rightarrow (LOCK = 0)$, and this is the proof obligation generated by the verification condition generator. We break this obligation into obligations for each internal node, so we need to prove the two obligations $(LOCK = 0) \Rightarrow (LOCK = 0)$ and $\mathit{false} \Rightarrow (LOCK = 0)$. Each proof is discharged using a proof generating theorem prover, and the entire proof is now constructed by combining these proofs. The complete proof encoded in LF is:

$$\begin{array}{l}(\mathtt{alli}\ [LOCK\ :\ exp]\ (\mathtt{imp\text{-}dist}\\ \quad (\mathtt{impi}\ [H_1\ :\ \mathtt{pf}\ \mathit{false}](\mathtt{falsee}\ (=\ LOCK\ 0)\ H_1))\\ \quad (\mathtt{impi}\ [H_2\ :\mathtt{pf}\ (=\ LOCK\ 0)]\ (H_2)))).\end{array}$$

□

4.2.4 Experiments

The cost of verification and certification is dominated by the cost of theorem proving, so we incorporate automatic lemma extraction by caching theorem prover calls. Our experiments

Program	Postprocessed LOC	Proof Size (bytes)
qpmouse.c	23539	175
ide.c	18131	253
tlan.c	16506	405
cdaudio.c	17798	156787
floppy.c	17386	60129
kbfiltr.c	12131	7619
parport.c	61781	102967

Table 4.3: Proof Sizes. The first three rows are for Property 1, rest are are for property 2.

show that many atomic proof obligations that arise during the entire process are identical, and so the size of the proof in dag representation is considerably smaller than a derivation tree. While our running times and proof sizes are encouraging, we feel there is a lot of room for improvement. We expect, based on previous experience, that the use of an oracle-based representation of proofs [NR01], would further reduce the size of the proofs by an order of magnitude.

4.3 Tests from Counterexample Traces

The software engineer is often interested in the set of *all* program locations where the property may be violated: given a predicate p, the programmer may wish to know the set of all program locations q that can be reached such that p is true at q. For example, when checking the security properties of a program it is useful to find the locations where the program has root privileges. We now show how to extend Blast to provide this information. As a special case (take p to be the predicate that is always true), Blast can be used to find the reachable program locations, and by complementation, it can detect dead code.

Moreover, if Blast claims that a certain program location q is reachable such that the target predicate p is true at q, then from the program trace that exhibits p at q, the tool automatically produces a test vector that witnesses the truth of p at q. This feature enables the software engineer to pose reachability queries about the behavior of a program, and to automatically generate test vectors that satisfy the queries [Pel01]. Technically, we symbolically execute the counterexample trace produced by the model checker, and extract a satisfying assignment of the symbolic constraints as a test vector. In particular, for a

predicate p and its negation, the tool automatically generates for each program location q, if p is always true at q, a test vector that exhibits p at q; if p is always false at q, a test vector that exhibits $\neg p$ at q; and if p may be true or false at q, then two test vectors, one that exhibits the truth of p at q, and another one that exhibits the falsehood of p at q. In this way, BLAST generates more informative test suites than any tool that is purely based on coverage, because the program locations of the third kind are each covered by two test vectors with different outcomes.

Often a single test vector covers the truth of p at many locations, and the falsehood of p at others, and BLAST produces a small set of test vectors that provides the desired information. It is essential that BLAST uses incremental model checking technology [HJMS03], which reuses partial proofs and counterexamples as much as possible. We have used our extension of BLAST to query C programs with 30 K lines of code about locking disciplines, security disciplines, and dead code, and to automatically generate corresponding test suites.

Related Work. There is a rich literature on test-vector generation using symbolic execution [Cla76; Kin76; RHC76; JBW+94; GMS98; GBR02; KPV03]. Our main insight is that given a particular target, one can guide the search to the target efficiently by searching only an *abstract* state space, and refining the abstraction to prune away infeasible paths to the target found by the abstract search. This is exactly what the model checker does for us. In contrast, unguided symbolic-execution based methods would have to precisely execute many more paths, resulting in scalability problems. Therefore, most research on symbolic-execution based test generation curtails the search by bounding, *e.g.* the number of iterations of loops, or the size of the input domain [JV00; BKM02; KPV03]. Unfortunately, this makes the results incomplete: if no trace to the target is found, one cannot conclude that no execution of the program reaches the target. Of course, once a suitable trace to the target is found, all previous methods to generate test vectors still apply.

This is not the first attempt to use model checking technology for automatic test-vector generation. However, the previous work in this area has followed very different directions. For example, the approach of [HCL+03] considers fixed boolean abstractions of the input program, and does not automatically refine the abstraction to the degree necessary to

```
#include <stdlib.h>
#include <stdio.h>

int readInt(void);

int middle(int x, int y, int z) {
L1:   int m = z;
L2:   if(y < z)
L3:     if(x < y)
L5:       m = y;
L6:     else if(x < z)
L9:       m = x;
      else
L10:    if(x > y)
L12:      m = y;
L13:    else if(x > z)
L15:      m = x;
L7:   return m;
}
int main() {
  int x, y, z;
  printf("Enter the 3 numbers: ");
  x = readInt();
  y = readInt();
  z = readInt();
  printf("Middle number: %d", middle(x,y,z));
}
```

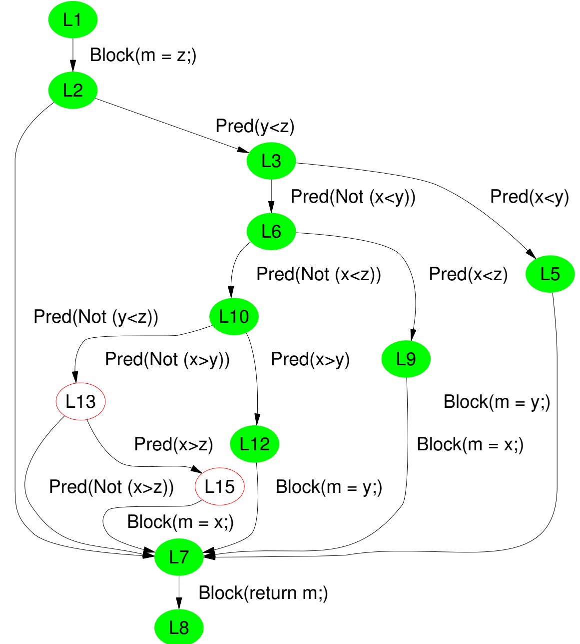

Figure 4.5: `middle` (a) Program (b) CFA

generate test vectors that cover all program locations for a given set of observable predicates. Peled [Pel03] proposes three further ways of combining model checking and testing. Black-box checking and adaptive model checking assume that the actual program is not given at all or not given fully. Unit checking [GP02] is the closest to our approach in that it generates test vectors from traces, however, these traces are not found by automatic abstraction refinement.

Organization. We begin with an overview in Section 4.3.1. Then, in Section 4.3.2 present the testing framework, and in Section 4.3.3 we show how test vectors are generated from counterexamples, how sufficiently many counterexamples are obtained to guarantee coverage for the resulting test suite, and how the corresponding test driver is constructed from the program. In Section 4.3.4 we conclude by presenting some applications and experimental results.

4.3.1 Overview

We first give an overview of the method using a few small examples. Consider the program of Figure 4.5(a), which computes the middle value of three integers. The program takes three inputs and invokes the function `middle` on them. A test vector for this program is a

triple of input values, one for each of the variables `x,y,z`. The right column of Figure 4.5(b) shows the control-flow automaton (CFA) for `middle`. The CFA is essentially the control-flow graph of `middle` with the control locations as nodes, and edges labeled by the operations that take the program from one node to the next —either basic blocks of assignments, or predicates that correspond to branch conditions which must be true for control to flow across an edge. For brevity, we omit the CFA for the `main` function. We first consider the problem of location coverage, *i.e.,* we wish to find a set of test vectors such that for each location of the CFA, there is some test vector in the set that drives the program to that location.

Phase 1. Model checking. To find a test vector that takes the program to location `L5`, we first invoke BLAST to check the property that `L5` is reachable. BLAST proceeds by iterative abstraction refinement to check that `L5` is reachable and, if this is the case, it finds a counterexample, *i.e.,* a trace to `L5` in the CFA. This trace is given by the following sequence of operations: `m=z; assume (y<z); assume (x<y);` where the first operation corresponds to the assignment upon entry, and the second and third (`assume`) operations correspond to the first two branch conditions being taken.

Phase 2. Tests from counterexamples. In the second step, we use the counterexample trace from the model-checking phase to find a test vector, *i.e.,* an initial assignment for `x,y,z` that takes the program to location `L5`. This is done as follows. First, we build a trace formula (TF), which is a conjunction of constraints, one constraint per operation in the trace. In this case the formula is $(m = z) \wedge (y < z) \wedge (x < y)$. Second, the feasibility of the trace implies that the TF is be satisfiable, and we find a satisfying assignment to the formula, *e.g.* "`x=0,y=1,z=2,m=2`", which after ignoring the value for `m`, gives a test vector that takes the program to `L5`.

We repeat these two phases for each location, noting that one input takes us to several locations —those along the trace— until we have a set of test vectors that covers all locations of the CFA. Along with each test vector, BLAST also produces a trace in the CFA that is exercised by the test. A set of test vectors for node coverage of `middle` is shown in Figure 4.6. Each row in the table gives an input vector —initial values for `x,y,z`— and the corresponding trace as a sequence of locations. For example, the vector of test values for

the target location L12 is (1,0,1), and BLAST reports the trace ⟨L1,L2,L3,L6,L10,L12⟩, which is easy to understand with the help of the CFA in Figure 4.5(b). The trace is a prefix of the complete program execution for the corresponding test vector.

The alert reader will have noticed that the tests do not cover all locations; L13 and L15 remain uncovered, as denoted by the absence of shading in Figure 4.5(b). It turns out that BLAST proves that these locations are not reachable —*i.e.,* they are not visited for any initial values of x,y,z— and hence there exists dead code in middle. A close analysis of the source code reveals that a pair of braces is missing, and that the indentation is misleading for the code without braces: the if on L6 matches the else after L9, which is meant for the if on L2.

Executing tests. To execute the generated tests, we automatically build a test driver from the given program. We feed the program and the name of the initial function from which the program's execution begins, into BLAST 's test-driver generator, which results in a C program that is compiled into a test driver. The test driver reads a file containing a set of test vectors we wish to run, and executes the program being tested using the vectors as input values. The user may run the driver in a debugger to study the dynamic behavior of the program under the various test inputs.

A security example. We now show how BLAST can offer help to the programmer to check for security vulnerabilities in programs. Figure 4.7 shows a simple program that manipulates Unix privileges using setuid system calls. Unix processes can execute in several privilege levels; higher privilege levels may be required to access restricted system resources. Privilege levels are based on process user id's. Each process has a real user id, and an effective user id. The seteuid system call is used to set the effective id, and hence the privilege level of a process. The user id 0 (or root) allows a process full privileges to access all system

x	y	z	Counterexample Trace
0	0	0	⟨L1,L2,L7,L8⟩
0	1	2	⟨L1,L2,L3,L5⟩
0	0	1	⟨L1,L2,L3,L6,L9⟩
1	0	1	⟨L1,L2,L3,L6,L10,L12⟩

Figure 4.6: Generated test vectors for middle

```
int saved_uid, saved_euid;
int get_root_privileges () {
L1: if (saved_euid!=0) {
L2:   return -1;
    }
L3: seteuid(saved_euid);
L4: return 0;
}
work_and_drop_priv() {
L5: FILE *fp = fopen(FILENAME,"w");
L6: if (!fp) {
L7:   return;
    }
L8:   // work
L9: seteuid(saved_uid);
}
```

```
int main(int argc, char *argv[]) {
L10:saved_uid = getuid();
L11:saved_euid = geteuid();
L12:seteuid(saved_uid);
L13:  // work under normal mode
L14:if (get_root_privileges ()==0){
L15:  work_and_drop_priv();
    }
L16:execv(argv[1], argv + 1);
}
```

Figure 4.7: The `setuid` example program

resources. We assume for our program that the real user id of the process is not zero, *i.e.,* the real user does not have root privileges. This specification is a simplification of the actual behavior of `setuid` system calls in Unix [CWD02], but is sufficient for exposition.

The `main` routine first saves the real user id and the effective user id in the variables `saved_uid` and `saved_euid`, respectively, and then sets the effective user id of the program to the real user id. This last operation is performed by the function call `seteuid`. The function `get_root_privileges` changes the effective user id to the id of the root process (id `0`), and returns `0` on success. If the effective user id has been set to root, then the program does some work (in the function `work_and_drop_privileges`) and sets the effective user id back to `saved_uid` (the real user id of the process) at the end (`L9`). To track the state changes induced by the `setuid` system calls, we instrument the code for the relevant system calls as follows. The user id is explicitly kept in a new integer variable `uid`; the `getuid` function is instrumented to return a nonzero value (modeling the fact that the real user id of the process is not zero); and the `geteuid` function is instrumented to nondeterministically return either a zero or a nonzero value. Finally, we change the `seteuid(x)` system call so that the variable `uid` is updated with the argument `x` passed to `seteuid` as a parameter. The instrumented versions are omitted for brevity.

Secure programming practice requires that certain system calls that run untrusted programs should not be made with root privileges [CW02], because the privileged process has

```
L10: saved_uid = getuid();
  /* body of getuid omitted */
L11: saved_euid = geteuid();
  /* body of geteuid omitted */
  /* geteuid returns 0 */
L12: seteuid(saved_uid);
  /* uid = saved_uid */
L14: tmp = get_root_privieleges();
  L1: if (saved_euid!=0) /* fails */
  L3: seteuid(saved_euid);
  /* uid = saved_euid */
  L4: return 0;
L14: if (tmp==0) /* succeeds */
L15: work_and_drop_priv();
  L5: fp = fopen(FILENAME, ‘‘w’’);
  L6: if (!fp) /* succeeds */
  L7: return;
L16: /* uid = 0 */
```

Figure 4.8: A trace generated by BLAST

full permission to the system. For example, calls to `exec` and `system` must never be made with root privileges. Therefore it is useful to check which parts of the code may run with root privileges.

We use the model checker BLAST in test mode to check which code lines can be executed with root privileges. More specifically, we ask the model checker to output all locations that are reachable in the program, with `uid=0` as target predicate (which indicates root privileges). For each such location, BLAST generates a test vector that causes the program to reach that location with the system being in a state where `uid=0`.

The BLAST output shows, surprisingly, that the `execv` system call can be executed with root privileges. An inspection of the symbolic program trace generated by the model checker (Figure 4.8) shows that there is a bug in the `work_and_drop_privileges` function: if the call to `fopen` fails, the function returns without dropping root privileges.

4.3.2 Testing Framework

Testing is usually carried out within a framework comprising (1) a suitable representation of the program, (2) a representation for test vectors, and a set of test vectors called a *test suite*, (3) an *adequacy* criterion that determines whether a test suite adequately tests the program, (4) a test generation procedure that generates an adequate test suite, and (5) a

test driver that executes the program with a given test vector by automatically feeding input values from the vector.

Programs and tests. We use CFAs as our representation of programs. This representation is very similar to the control-flow graphs [ASU86] used in many testing frameworks. A *test vector* is a sequence of input data required for a single run of the program. This sequence contains the initial values for the formal parameters of the program, and the sequence of values supplied by the environment whenever the program asks for input. In other words, in addition to input values, the test vector also contains a sequence of return values for external function calls. For example, when testing device drivers, the test vector would contain a sequence of suitable return values for all calls to kernel functions made by the driver, and a sequence of values for data read off the device.

Target predicate coverage. Ideally, one would like the test suite to exercise all execution paths of the program ("path coverage"), and thus expose any errors that the program may have. As such test suites will be infinitely large for most programs, the notions of location and edge coverage are used to approximate when a program has been tested sufficiently[Mye79; PY03].

We use the following notion of *target predicate coverage*: given a C program in the form of a CFA, and a target predicate φ, we say a test vector *covers* a location pc of the CFA w.r.t. φ if the execution resulting from the vector takes the program into a state where it is in location pc and the variables satisfy the predicate φ. We deem a test suite adequate w.r.t. φ if all φ-reachable CFA locations are covered w.r.t. φ by some vector in the suite.

For example, consider the program in Figure 4.7 and the target predicate `uid=0`. The algorithm outputs tests vectors for all locations that the program can reach with the value of `uid` being `0`. As another case, suppose that the target predicate φ is *true*. Then the test-generation algorithm outputs test vectors for all *reachable* CFA locations. Furthermore, BLAST reports all CFA locations that are (provably) unreachable by any execution, as dead locations (they correspond to dead code). If we run BLAST on a program with both predicates φ and $\neg\varphi$, then for all CFA locations pc that can be reached with φ either true or false, we obtain *two* test vectors —one that causes the program to reach pc with φ true, and another one that causes the program to reach pc with φ false.

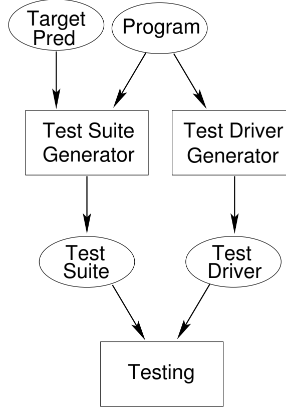

Figure 4.9: Test flow

The notion of target predicate coverage corresponds to *location coverage* ("node coverage") if $\varphi = true$. For *edge coverage*, for an edge e that represents a branch condition p_e, we can find a test that takes the program to the source location of e with the state satisfying the predicate p_e, thus causing the edge e to be traversed in the subseqent execution step. We can similarly adapt our technique to generate tests for other testing criteria [HCL+03; PY03]; we omit the details.

Test flow. The overall testing framework as implemented in BLAST is shown in Figure 4.9. A program and a target predicate are fed as inputs. The *test-suite generation* procedure takes the program and the target predicate as input and produces an adequate test suite, *i.e.*, one such that every φ-reachable CFA location is covered w.r.t. φ by some test vector in the suite. The *test-driver generation* procedure takes the program as input and produces another program, the test driver, that runs the original program on the test inputs. The test driver has a wrapper function for all external function calls, which returns values from the test vector to the program.

In the following subsections we describe how to generate an adequate test suite, and how to generate a test driver that can execute the program on the test vectors in the suite in order to allow developers to see how the program behaves on the generated tests.

4.3.3 Test Suite Generation

For ease of exposition, we describe our method only for programs (and CFAs) without function calls; it can be extended to handle function calls in a standard way (and function

calls are handled by the BLAST implementation).

Let $C = (X, PC, pc_0, \rightarrow)$, φ be a formula in $\mathsf{Pred}.X$ and $pc_{\mathcal{E}}$ be a special *error location* in PC. We say that a trace σ is a φ-trace to pc if σ is a counterexample to $(pc, (\varphi, \emptyset), \epsilon)$ from $(pc_0, (\mathit{true}, \emptyset), \epsilon)$. Recall that σ is a φ-trace to pc iff σ corresponds to a path to pc from pc_0 in the CFA, and, $\mathsf{SP}.\mathit{true}.\sigma \wedge \varphi$ is satisfiable. We say that a location pc is φ-reachable if there is a φ-trace to pc.

From the definitions, and Proposition 2 it follows that, if T is a reachability tree for C that is safe w.r.t. (pc, φ) from (pc_0, true), then pc is not φ-reachable in C.

In the sequel, assume that Algorithm 2 is modified so that it takes as input a CFA C (from which it builds a symbolic abstraction structure), a partial reachability tree (T, F), and a pair of (pc, φ), where pc is a target location and φ a target predicate. Upon termination, the modified algorithm returns with one of two outcomes: either O_1, a complete reachability tree T' that is safe w.r.t. (pc, φ) from (pc_0, true), or O_2, a partial reachability tree (T', F') that has a path from the root node $\mathtt{n}_0 \xrightarrow{\sigma} \mathtt{n}$ such that σ is a is a φ-trace to pc. In the former case, from Theorem 3 we conclude that pc is not φ-reachable in C; in the latter case, we shall extract from the program trace a test vector that drives the program to the location pc such that at pc, the program variables satisfy the predicate φ. Given a program and a target predicate φ, the test-suite generation now proceeds as follows.

Step 1. The locations of the CFA are numbered in depth-first order, and put into a *worklist* in decreasing order of the numbering (*i.e.,* the location numbered last in DFS order is first on the worklist). We create an initial partial reachability tree (T, F), where T is a tree with a single node, namely the root $\mathtt{n}_0 : (pc_0, \mathit{true})$, and F is the singleton set containing $\mathtt{n}_0$. The initial test suite is the empty set.

Step 2. If the worklist is empty, then we return the current test suite; otherwise let pc be the first CFA location in the worklist. We invoke the model checker with the partial reachability tree (T, F) and (pc, φ), and we update the current reachability tree with the result of the model checking.

Step 3. If the model checker returns with outcome O_1, then we conclude that for all locations pc' such that the new reachability tree (T', F') is safe w.r.t. (pc', φ), no test

vector exists, and so we delete all such locations from the worklist. Otherwise, if the model checker returns with outcome O_2, then we have a φ-trace to the location pc. We use this trace to compute a test vector that covers the location pc w.r.t. φ using a procedure described below. We add this vector to the test suite, and remove pc from the worklist. In both cases, we go back to **step 2**.

It can be shown that upon termination, the above procedure returns a test suite that is adequate w.r.t. φ according to our criterion of target predicate coverage.

We incorporate several optimizations to the above loop. First, when a test vector is found, we can additionaly find (by symbolically executing the program on the vector) which other locations it covers, and we remove those locations from the worklist. Second, the model-checking algorithm uses heuristics to choose the next node to unfold in the partial reachability tree. The nodes that need to be unfolded are partitioned into those that have been covered by a vector in the current test suite, and those that are still uncovered. The model checker unfolds uncovered nodes first, and it unfolds covered nodes only if there remain no uncovered nodes. A node that has been covered by a previous test may still need to be unfolded, because a path to an (as yet) uncovered location may go through it. Third, the user has the option to give a time-out for the model checking. Thus in **step 3**, if instead of O_1 or O_2, the model checker times out, then we give up on the location pc, by deleting it from the worklist and going back to **step 2**. We have found these optimizations to be essential for the algorithm to work on large programs.

Generating tests from traces

When model checking in **step 2** ends with outcome O_2, the resulting tree contains a path to a node $\mathtt{n} : (q, r)$ such that the path corresponds to a φ-trace ending at pc. We now describe how to extract from this trace a test vector that, when fed to the program, takes it to location pc satisfying the target predicate φ.

Constraint generation. Recall that to check if a trace σ corresponding to a path in the tree is feasible, we check if the $\mathsf{SP}.true.\sigma$ is satisfiable. To build a test vector from the trace, we shall first construct the Trace Formula for the trace using the operator Con described in Section 3.3. As the trace is feasible, the TF is satisfiable (from Theorem 5). We use a

(a) Program	(b) Trace	(c) Trace formula	(d) Assignment	(e) Test vector
`Example() {`				
`if (y == x)`	`assume (y=x)`	$\langle y,0\rangle = \langle x,0\rangle$	$\langle x,0\rangle \mapsto 0$	$x \mapsto 0$
`y ++ ;`	`y = y+1`	$\langle y,1\rangle = \langle y,0\rangle + 1$	$\langle y,0\rangle \mapsto 0$	$y \mapsto 0$
`if (z <= x)`	`assume !(z<=x)`	$\neg\langle z,0\rangle \leq \langle x,0\rangle$	$\langle y,1\rangle \mapsto 1$	$z \mapsto 2$
`y ++ ;`			$\langle z,0\rangle \mapsto 2$	
`a = y - z;`	`a = y-z`	$\langle a,2\rangle = \langle y,1\rangle - \langle z,0\rangle$	$\langle a,2\rangle \mapsto -1$	
`if (a < x)`	`assume (a<x)`	$\langle a,2\rangle < \langle x,0\rangle$		
`LOC:`				
`}`				

Figure 4.10: Generating a test vector

decision procedure to produce a satisfying assignment for the variables of the TF. From the satisfying assignment we build a test vector that drives the program to the target location and target predicate. Given a φ-trace σ to pc, Let $(\theta, \Gamma) = \mathsf{Con}.(\theta_0, \Gamma_0 :: \mathit{true}).\sigma$. Then, the TF is $\bigwedge_{|\sigma|} \Gamma \wedge \mathsf{Sub}.\theta.\varphi$.

Tests from constraints. The TF is a conjunction of constraints about special constants of the form $\langle l, i\rangle$, each of which is an arithmetic fact that relates the values of program variables at various points in the trace. In our experience, many programs generate *linear* arithmetic constraints. Thus, we can find a satisfying assignment for the TF using an integer linear programming (ILP) solver. For a satisfiable formula φ, let $S.\varphi$ be a satisfying interpretation of all special constants that occur in the formula. A test vector that exercises the trace t is obtained by setting every input variable x of the program to the initial value $S.\varphi.\langle x, 0\rangle$.

Example 11 [Traces, Tests] Figure 4.10(a) shows a program, and Figures 4.10(b) and 4.10(c) show, respectively, a trace to the program location `LOC`, and the TF for that trace. The constraint for each atomic operation of the trace is shown to the right of the operation; the TF is the conjunction of all constraints. Figure 4.10(d) shows a satisfying interpretation for the special constants of the TF of Figure 4.10(c). It is easy to check that if we set the inputs initially to "`x=0, y=0, z=2`," then the program follows the trace of Figure 4.10(b). The generated test vector is shown in Figure 4.10(e). □

Pointers. The above method can be extended to programs with pointers. We first generate

the TF from whose satisfying assignment we obtain a test vector as described above; the details of the TF generation are given in in [HJMM04]. The resulting TF contains disjuncts due to possible aliasing. There are two ways to deal with this. First, we can convert the formula to DNF and check each disjunct separately, and on finding a satisfiable disjunct, we can extract a test vector from a satisfying assignment of the disjunct. Second, we can use efficient decision procedures for propositional satisfiability [MMZ+01] to find a possibly satisfiable disjunct, and then use the ILP solver to find a satisfying assignment for that disjunct, from which again the tests are computed as discussed above. Many off-the-shelf decision procedures already incorporate this style of propositional reasoning [FORS01; SBD02; HJMM04]. Of course, there are programs for which our constraint-based test-generation strategy fails because the given constraint language is not expressive enough.

Library calls. If a trace contains library calls whose source code is not available for analysis, or asks for user input, the constraint generation assumes that the library call or the user can return any value. Thus, some of our tests may not be executable if the library calls always return values from some subset of possible values. In this case, the user can model postconditions on library calls by writing stub functions that restrict the possible return values.

Test Driver Generation

Recall that a test vector generated by Blast is a sequence of integer values (our test-vector generation is currently restricted to integer inputs): these are the values that are fed to the program by the test driver during the actual test; they include the initial values for all formal parameters and the return values for all external function calls.

The test-driver generator takes as input the original program and instruments it at the source-code level to create a test driver containing the following components: (1) a wrapping function, (2) a test-feeding function, and (3) a modified version of the original program. The test driver can then be compiled and run to examine the behavior of the original program on the test suite. It can be run on each individual test vector and the user can study the resulting dynamic behavior as she pleases, by using a debugger for example.

The wrapper is the main procedure of the driver: it reads a test vector and then calls

Table 4.4: Experimental results

Program	LOC	CFA locations	Locations			Tests	Predicates		Time
			Live	Dead	Fail		Total	Average	
kbfiltr	5933	381	298	83	0	39	112	10	5 min
floppy	8570	1039	780	259	0	111	239	10	25 min
cdaudio	8921	968	600	368	0	85	246	10	25 min
parport	12288	2518	1895	442	181	213	509	8	91 min
parclass	30380	1663	1326	337	0	219	343	8	42 min
ping	1487	814	754	60	0	134	41	3	7 min
ftpd	8506	6229	4998	566	665	231	380	5	1 d

the main function of the original program, passing it initial values for the parameters from the vector. The driver generator modifies the code of the program being tested by replacing every call to an external function with a call to the special test-feeding function. The test-feeding function reads the next value from the test vector and returns it. We are guaranteed that the vector will have taken the program to the target when the test-feeding function has consumed the entire vector. Hence, once the test vector is consumed, the feeder returns arbitrary values.

4.3.4 Experiments

We ran Blast to generate test suites for several programs. We used two sets of benchmark programs: a set of Microsoft Windows device drivers, and two security-critical programs. The results are summarized in Table 4.4. The programs `kbfiltr`, `floppy`, `cdaudio`, `parport`, and `parclass` are Microsoft Windows device drivers. The program `ping` is an implementation of the ping utility, and `ftpd` is a Linux port of the BSD implementation of the ftp daemon. The experiments were run on a 3.06 GHz Dell Precision 650 with 4 GB of memory.

We present results for checking the reachability of code. In these experiments, the specification was trivial (*i.e.,* the target predicate was *true*): we checked which program locations are *live* (reachable by some execution) and *dead* (not reachable by any execution), and we generated test vectors that cover all live locations. Syntactically plausible executions (for example, control-flow paths, or data flows) may not be semantically possible, for example, due to correlated branching [BGS97]. This is called the *infeasibility problem* in testing [PY03; JBW+94]. The usual approach to deal with infeasible paths is to argue manually on a case-

by-case basis, or to resort to adequacy scores (the percentage of all static paths covered by tests). By using Blast we can automatically detect dead code, and generate tests for live code.

In the table, *LOC* refers to lines of code. CFAs represent programs compactly; each basic block is a single edge. In the table, the column *CFA locations* shows the number of locations of the CFA which are syntactically reachable by exploring the corresponding call graph of the program. *Live* is the number of reachable locations, *Dead* is the number of unreachable locations, and *Fail* is the number of locations on which our tool failed. Ideally, the total number of CFA locations is equal to the sum of the live and dead locations. However, in our tool we set a time-out for each location. So in practice, the tool fails on a small percentage of locations. The failure is due both to time-outs, and to not finding suitable predicates for abstraction. In our experiments, we set the time-out to 10 minutes per location.

The column *Tests* gives the number of tests generated. The implementation does not run the model checker for a location that is already covered by a previous test. Thus, the number of tests is usually much smaller than the number of reachable locations. This is especially apparent for the larger programs. *Total* is the total number of predicates, over all locations, generated by the model-checking process. *Average* is the average number of predicates active at any one program point. The average number of predicates at any location is much smaller than the total number of predicates, thus confirming our belief that local and precise abstractions can scale to large programs [HJMS02; HJMM04]. *Time* gives the running time rounded to minutes (except for `ftpd`, where the tool ran for two overnight runs).

We found many locations that were not reachable because of correlated branches. For example, in `floppy`, we found the following code:

```
driveLetterName.Length = 0;
// cut 15 lines
...
if (driveLetterName.Length != 4 ||
  driveLetterName.Buffer[0] < 'A' ||
```

```
    driveLetterName.Buffer[0] > 'Z') {
      ...
  }
```

Here, the predicate `driveLetterName.Length != 4` is true; so the other tests are never executed. Another reason we get dead code is that certain library functions (like `memset`) make many comparisons of the size of a structure with different built-in constants. At run time, most of these comparisons fail, giving rise to many dead locations.

While the table reports only experiments that check for unreachable code, we ran Blast also on several small examples with security specifications in order to find which parts of a program can be run with root privileges. Unfortunately, most security programs make recursive calls, and our previous implementation of Blast did not support recursive function calls. We are currently implementing a new version that does handle recursive calls. We are also optimizing our test-generation procedure to generate tests directly from the internal data structures of the model checker.

Chapter 5

Multithreaded Programs: Context Inference

We now turn our attention to the safety verification problem for concurrent, multithreaded programs. Dynamic methods, *e.g.* testing, are particularly inadequate in the multithreaded setting as the combinatorial explosion of the possible number of behaviours arising from the many ways that single threaded executions can be interleaved by schedulers, makes erroneous executions difficult to find and reproduce.

A classic safety verification problem for concurrent programs is *data race detection*: a data race occurs when two threads can access (read or write) a data variable simultaneously, and at least one of the two accesses is a write. The program is race-free if no such state is reachable.

Consider, for example, the "test-and-set" nesC program taken from [GLvB+03] in Figure 5.1. Previous approaches, which require that *locks* be the mechanism by which race-freedom is enforced, falsely flag this program as potentially buggy, as it uses the *value* of the variable `state` instead of explicitly declared locks to guarantee race-freedom. The first thread to enter the atomic block sets its local variable `old` to 0 and the global `state` to 1, and gets to access `x`. The other threads copy the value 1 into their local copy of `old`, and the check `old` $= 0$ before accessing `x` precludes the possibility of a race on `x`. In many programs, the problem is harder as the accesses to the "protected" variable happen in procedures other than the ones where the variable `state` is toggled, and often happen only if the function that changes the "state" variable returns a particular value ("conditional locking"). Other synchronization mechanisms, such as the enabling and disabling of certain interrupts, are

also beyond the scope of methods based on locks. A more precise path and interleaving sensitive analysis, such as model checking, that tracks the values of variables is required to verify the absence of races.

The difficulty with directly model checking multithreaded programs arises from the interleaving of executions of concurrent threads which causes an exponential explosion in the control state that must be tracked. In programs where threads can be dynamically created, the problem is worse as the set of control states is unbounded.

Our approach to the safety verification problem for multithreaded programs is to consider the system as comprising a *main* thread and a *context* which is an abstraction of all the other threads in the system, and then verifying (a) that this composed system is safe ("assume") and (b) that the context is indeed a *sound* abstraction ("guarantee"). Once the appropriate context has been divined, the above checks can be discharged by existing methods [God97; CDH+00; Hol00; HP00; FQS02]. Additionally, the remaining data abstraction can be performed automatically using counterexamples [BR00; HJMS02; COYC03].

If the context is imprecise, then either of the above checks may fail, leaving us with no information about the safety of the multithreaded program. This poses the following questions: First, what is a model for the *context* that is simultaneously (i) abstract enough to permit efficient checking and (ii) precise enough to preclude false positives as well as yield real error traces when the checks fail. Second, how can we infer such a context automatically.

We were tempted to answer these questions as follows [HJMQ03]. We chose as context model, a *relation* R on the global variables, which represents the possible effects that the other threads may have on the global state between any two transitions of the *main* thread, *i.e.,* at any point, the context could change the global variables from s to s' so long as $(s, s') \in R$. We designed an algorithm to infer such contexts using CEGAR.

Experiments showed that this *stateless* context model lacks the precision required to prove the safety of programs such as the ones described earlier, and to produce error traces for buggy programs. As context threads change the global variables depending on their local states, statelessness leads to false positives. Also, to generate error traces (and to refine abstractions) we must be able to check if an abstract trace corresponds to *some* concrete

interleaving of the program's threads. This is difficult if the context has no information about the other threads' local states. For these reasons, the context must track the *local state* of its threads. Unfortunately, statefulness brings the burden of tracking the state of *each* of the arbitrarily many context threads. We now present a richer model for contexts that solves both the above problems, and a generalization of the algorithm from [HJMQ03] that constructs these richer context models automatically.

Stateful Contexts. First, we represent each context thread by an *Abstract Thread* (AT). Each AT location corresponds to a set of control locations of the thread, and we keep ATs minimal by computing weak bisimilarity quotients [CGP99]. Each AT location is labeled by a formula over the globals, which constrains the possible values of the global variables. Second, we track the state of each of arbitrarily many context ATs by labeling each AT location with an integer counter (possibly ω), which represents the *number* of threads at that location ("counter AT"). Thus, our context models combine three forms of abstraction: predicates for data abstraction, weak bisimilarity quotients for control abstraction, and counters for abstracting multiple threads.

Context Inference. Suppose, for simplicity, that all threads run the same code as *main*. In general, our algorithm requires that each of the threads be running one of finitely many pieces of code, and that the threads do not reference each other. The inference of context models proceeds in two nested loops. The outer loop sets the context model to be the *strongest* model (which does not interfere with *main*) and then executes the inner loop. Given a predicate abstraction of *main*, and a counter AT that represents the multithreaded context, the inner loop iteratively weakens the context model until either (i) an abstract error is found, or (ii) the resulting counter AT overapproximates (simulates) the program. If (i) happens, we break out of the inner loop and analyze the abstract counterexample. If it is real we report the bug and exit, if it is spurious we add new predicates or refine the counter, and repeat the outer loop. If (ii) happens, we conclude (by assume-guarantee reasoning) that the program is free of races and exit. Otherwise (*i.e.,* neither (i) or (ii)), we weaken the context model by transforming the current reach set of *main* into a new AT and repeat the inner loop with the new, weaker context model. The whole process stops when either a concrete race is found, or the absence of races is proved using a context which

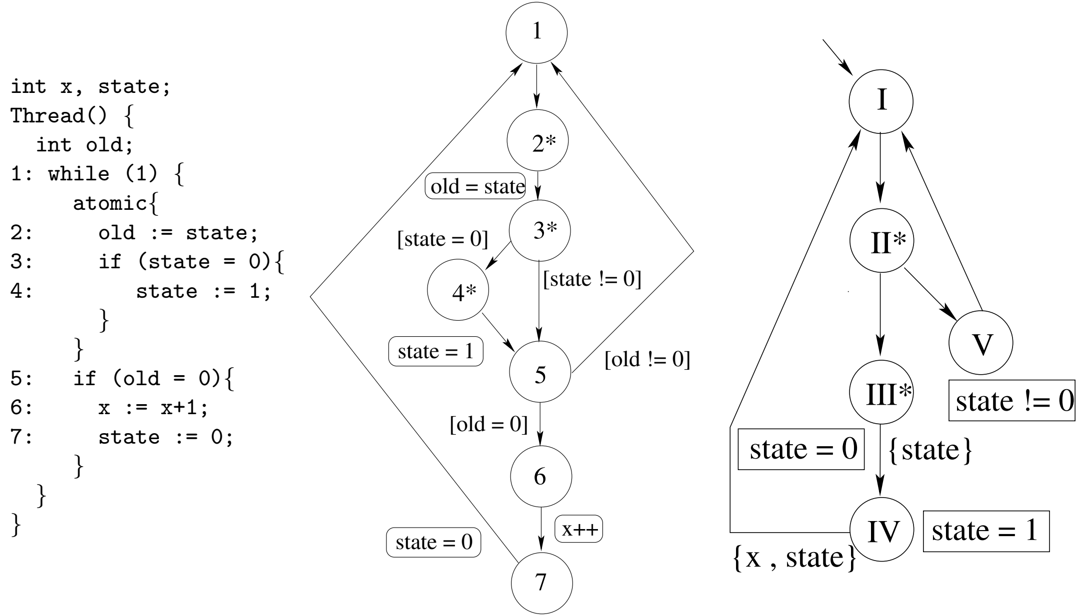

Figure 5.1: (a) Thread (b) CFA (c) AT

overapproximates the program.

While our method applies to verifying any safety property of concurrent programs, we have focused on race detection for two reasons. First, race checking requires no code annotations or specifications from the user. Second, the absence of race conditions is a prerequisite for establishing a variety of more complicated correctness requirements.

In the next Section 5.1, we present our method using the example from Figure 5.1. Then, in Section 5.2 we present our formal model of multithreaded programs and the approach of thread-context reasoning. In Section 5.3 we describe abstractions for multithreaded programs, and in Section 5.4 we describe how to use those abstractions to verify multithreaded programs using Thread-Context Abstraction-Refinement. In Section 5.6 we describe our experiences with looking for races in nesC programs, and we conclude the chapter in Section 5.7 with a result about the completeness of counter abstractions.

5.1 An Example

We begin by illustrating our method using an example. The formal development is postponed to the next section. Consider the fragment of code shown in Figure 5.1, taken from a nesC program [GLvB+03]. This fragment describes the behavior of a single thread; `x`

and `state` are global variables and each thread has a local variable called `old`. The multithreaded program P has an arbitrary number of threads running this code concurrently. We wish to verify that there are no races on `x` in P, *i.e.,* that P never reaches a state where two (or more) threads are about to access (read or write) `x`, and one of the accesses is a write.

5.1.1 Threads

Threads. We represent each thread as a *Control Flow Automaton* (CFA) (cf. Section 2.2.1). Our method and implementation handles threads comprising a set of (non-recursive) functions, but for clarity, we shall consider each thread to have a single CFA.

Example 12 [Thread] Consider the CFA shown in the middle in Figure 5.1 for the thread shown on the left in the same figure. The vertices marked with $*$ are atomic locations. The `atomic` construct of NESC allows a sequence of operations to occur without preemption; atomic locations model this. If in a multithreaded program, a thread is at an atomic location, only that thread is allowed to execute. Additionally, we assume that each command labeling an edge executes without preemption. If not, the edge is broken into an appropriate sequence of edges. $\square$

Informal Semantics. A multithreaded program is a set of threads where each thread is represented by a CFA. A *state* of a multithreaded program is a valuation for all the variables, including the global variables shared by all threads and each thread's local variables (*e.g.* program counter). We shall assume for clarity that all threads have the same CFA. In the initial state, each thread is at the start location, and all the variables have value 0. The system evolves as follows. (1) A thread is scheduled: if some thread is at an atomic location, it gets to run, otherwise some thread is chosen non-deterministically. (2) The scheduled thread picks one of the out-edges of the location it is at and executes it and proceeds to the target of the edge. If the edge is an assume, this happens only if the state satisfies the predicate and the variables remain unchanged; if the edge is an assignment `x:=e` then the expression `e` is evaluated and written into `x`, and then the program moves to the target location of the edge. It can be checked that if the start location is not atomic, then in any reachable state at most one thread is at an atomic location.

Data Races. There is a *data race* on the variable `x` if the program can reach a state in which two or more threads have enabled actions that read or write `x`, and at least one of these accesses is a write. We say a thread can write (read) `x` if there is an out-edge from its location where `x` is assigned (read). Thus a state has a race on `x` if (1) no thread is at an atomic location, and (2) one thread is at a location where `x` may be written and another is at a location that may access `x`. In the program comprising threads of Figure 5.1, there are no races on `x` if in every reachable state, at most one thread is at location `6`.

5.1.2 Thread-Context Programs

We analyze a multithreaded program as a thread-context program (TCP), which comprises a *main* thread executing in a *context* which represents all the other threads. We model each of the context threads using an abstract thread.

Abstract Threads. An abstract thread (AT) is a directed graph, whose vertices are *abstract control locations* labeled by predicates on the global variables of the program, and optionally by *atomic*, and whose edges are labeled by sets of *havoced* global variables. When the automaton moves from one location to the next, the havoced variables on the traversed edge are written to with arbitrary values, but the successor state is constrained to satisfy the predicate labeling the successor location.

Example 13 [Abstract Thread] Figure 5.1(c) shows an AT for the thread of the example. Locations labeled $*$ are atomic, and if there is an (abstract) thread at an atomic location, then only that (abstract) thread is scheduled. Each location is also labeled by a predicate inside a box, locations not labeled explicitly have the label `true`. Note this abstraction captures the essence of the behavior of the thread: first, it enters the atomic block, then if `state` is 0, it havocs `state` subject to the constraint that `state` is 1 in the next state. It then proceeds to access `x`, as it will have set its `old` to 0, and then havocs `state` to any arbitrary value. Alternately, if `state` is not 0 when the thread entered the block, then it would set its `old` to a non-zero value and thus loop back without writing to `x` or `state`. $\square$

Informal Semantics. A TCP, written $\mathsf{TCP}.(C, A)$ is a set $\{C\} \cup A^{\omega}$ comprising a main thread, represented by a CFA C, and a context which is an arbitrary number of abstract

threads A. The semantics of a TCP are similar to that of a multithreaded program. At the initial location, the main thread is at the start location of C and each context thread is at the start location of A. At each time step, either the main or a context thread is scheduled, and the scheduled thread makes a transition according to one of the out-edges of its current location.

5.1.3 Verification by Abstraction

Our method works by analyzing a TCP which is an abstraction of the multithreaded program we wish to verify. A precise analysis of the reachable states of a multithreaded program must abstract the state space to counter the infinite data valuations as well as the exponential number of possible program location tuples. Accordingly, we present three orthogonal abstractions.

1. Data Abstraction. The number of valuations for the program variables is infinite. To deal with this, we use predicate abstraction [AM78; GS97b], where instead of tracking the exact values of variables, we track relationships between program variables captured by boolean formulae over a finite set of predicates over the variables. Any local variable in a predicate refers to the *main* thread's copy of the local variable.

2. Control Abstraction. The number of configurations of other *context* threads is exponential in the number of program locations of each thread. To ameliorate this exponential blowup, we represent each context thread as an *abstract thread* which is a state machine that (1) has fewer locations than, and (2) overapproximates the behavior of (*e.g.* simulates), the thread it represents. All the predicates labeling the AT vertices are over the globals; information pertaining to the local state of context threads is encoded in the AT location.

3. Counters. There may be an arbitrary number of context threads. To make our analysis sound in this setting, we must model the AT location of each of the arbitrarily many context threads. To do this, we track the *number* of abstract threads that are at each of the finitely many AT control location [Lub84]. Since this representation is infinite, we use a *counter abstraction*: we track the number precisely so long as it is less than or equal to a parameter k, and any number greater than k is abstracted to ω, meaning an arbitrary number of

threads is at that abstract control location.[1]

Abstract Multithreaded Programs

To abstract TCP, we shall use regions to represent sets of states. We use a set of support predicates Λ from the thread's commands to finite sets of support predicates, to abstract the main thread. To abstract A^ω, we shall use *counter maps* which indicate how many copies of A are at each location of A. To keep the number of these maps finite, we shall treat all values above a parameter k to be ω.

A region of $\mathsf{TCP}.(T, A)$ is the tuple $((pc, \varphi), \delta)$, where pc is the *main* thread's control location, φ is a boolean formula over the range of Π (local variables refer to the main thread's copy of the local variable), and δ is a map from A's locations to $\{0, \ldots, k, \omega\}$. The operations enabled at a region are the operations enabled at pc and at each location n of A s.t. $\delta.n > 0$, so long as none of the above mentioned locations is atomic, otherwise, the enabled operations are the operations enabled at the (single) atomic location.

In the initial region, the main thread is in the initial location of C, δ is ω for the initial abstract location, and 0 elsewhere, and φ is a formula stipulating that all the variables are 0. For an region $\hat{s} = ((pc, \varphi), \delta)$ and an operation op, the successor region $\mathsf{post}.\hat{s}.\mathsf{op} = ((pc', \varphi'), \delta')$ is computed as follows. If the operation is the main thread's operation, then pc' is the target of the CFA edge taken, φ' is the predicate abstraction (w.r.t. Λ) of the strongest postcondition of φ w.r.t. the operation (cf. Section 2.2.2), and $\delta' = \delta$. If it is a context AT moving across an abstract edge $n \to n'$, then $pc' = pc$, φ' is the predicate abstraction (w.r.t. Λ) of $(\exists y_1 \cdots y_k.\varphi) \wedge r'$ where $y_1 \cdots y_n$ are havoced on edge $n \to n'$ and $n' : r'$, and δ' maps n to $\delta.n - 1$, n' to $\delta.n' + 1$, and all other n'' to $\delta.n''$.

Abstract Reachability. We build the set of reachable regions by iterating post from the initial region until a fixpoint. We check if there are races by checking if any reachable state contains a race. If so, the reachability procedure returns an abstract error trace. We say that G is an *abstract reachability graph* (ARG) for $\mathsf{TCP}.(C, A)$ if it is an AT that overapproximates the behavior of C in $\mathsf{TCP}.(C, A)$. The reachability procedure also computes an ARG G for $\mathsf{TCP}.(C, A)$ which we use to guarantee the soundness of A. If A is an overapproximation of (*e.g.* can simulate) G then we know that A is sound. If not, a minimized

[1] Note: $k + 1 = \omega$, $\omega + 1 = \omega$, and $\omega - 1 = \omega$

version of G gives us a better abstraction of the individual threads, which we use in the subsequent analysis.

5.1.4 The Algorithm CIRC

Given a CFA C, and a global variable `x`, we wish to verify that in the multithreaded program comprising arbitrarily many copies of C running concurrently, there are no races on `x`. In addition, the user may supply an initial set of predicates Λ (the default is $\emptyset$), and an initial counter parameter k (the default is 1).

Initialization ("Initial context") Set the initial AT A to be the empty AT, *i.e.,* the context does nothing.

Step 1 ("Reachability: Assume") Assuming that the context is made of threads behaving as A, compute the set of abstract reachable states of C using the present set of predicates P. Simultaneously build an abstract *reachability graph* (ARG) which is an AT G overapproximating the behavior of C in the current context (Algorithm ReachAndBuild in Section 5.4).

Step 2 ("Counterexample analysis") Check if the reachable states computed above contain states with races on `x`. If there are no such states, **go to** step 3. Otherwise, check whether this trace is real by first generating a concrete sequence of interleaved thread operations (from the sequence of thread/AT operations) and then checking if the interleaved trace is feasible. The concretization of the AT trace is done using the ARG of which the AT is the minimized version. Hence, every sequence of AT operations, corresponds to a (possibly infeasible) path through the underlying CFA. If (a) it was not possible to generate the concrete trace as the counter was too low, increment k, (b) the concrete trace is infeasible, infer new predicates using a Refine procedure (cf. Section 3.3) and add them to the set of predicates Λ, (c) the concrete trace *is feasible* then **return** UNSAFE with the genuine error trace. Reset A to the empty context and **go to** step 1.

Step 3 ("Guarantee") Check that the A assumed in step 1 was sound by checking that it overapproximates G computed in step 1 (Algorithm CheckSim). If so, **return** SAFE,

else, set A to be the bisimulation minimization of G (Algorithm Collapse), and **go to** step 1.

Running CIRC. We shall now run the algorithm on the example of Figure 5.1. Recall that there is no race on `x`. The first thread that goes inside the `atomic` block sets `state` to 1 and subsequent threads always set their `old` to 1 and so do not write `state` or `x`. Once the original thread has set `state` back to 0 the other threads can make another attempt, in which they set their `old` to 0, set `state` to 1 and then access `x`.

Initialization The initial AT A_0 is set to be the empty AT. The initial set of predicates Λ_0 is empty, but control flow is explicitly tracked.

Iteration 1

Step $1_1, 2_1$ The ARG G_1 of ReachAndBuild is shown in Figure 5.2(a). All the control locations are reachable and the state is just *true*, *i.e.,* we know nothing about the values of the variables of C. The reachability is trivially free of races as the context threads do nothing.

Step 3_1 Since A_0 was empty, Algorithm CheckSim detects that A_0 does not overapproximate G_1 and hence A_0 is unsound. Thus, we minimize G_1 to get the new AT A_1 shown in Figure 5.2(b). The dotted circles denote the sets of G_1 states that are merged into a single A_1 state. The minimized AT starts at a non-atomic location, then moves into an atomic location, in which it havocs `state` and moves to a non-atomic location from which it again havocs $\{$`x`, `state`$\}$ and returns to the start location. The locations `I,II` are not collapsed together as we wish to preserve atomicity, the same holds for `II,III`. Locations `I,III` are not collapsed as `x` can be written only in `III`. We repeat the loop setting A to be A_1.

Iteration 2

Step 1_2 On redoing reachability assuming the context threads behave as A_1 we find a race where one of the context threads moves two steps to reach the abstract location `III` (Figure 5.2(b)), following which the main thread moves to the concrete location `6`.

Step 2_2 We concretize the abstract trace described above and find that the thread followed an infeasible path: `1` $\rightarrow$ `2` $\rightarrow$ `3` $\rightarrow$ `5` $\rightarrow$ `6`, *i.e.,* the trace is infeasible without even considering the other thread. From this trace, we learn the predicates $old = state$ and $old = 0$ are

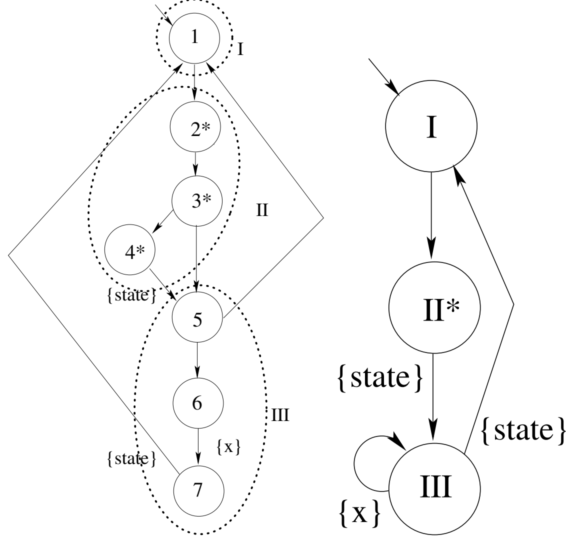

Figure 5.2: (a) ARG G_1 (b) Min. ARG A_1

required to rule out this infeasible path. We add these to get the new set of predicates Λ_2, set the context AT A_2 to be the empty AT, and go back to step 1.

Iteration 3

Steps $1_3, 2_3, 3_3$ We repeat the reachability using A_2 and Λ_2, to get the ARG G_3, shown in Figure 5.3(a). Notice that this time, the only path to the location where the write is enabled is a feasible path for each thread. Again, the reach set is trivially error free. As G_3 is not overapproximated by A_2, the latter being the empty AT, we set A to be A_3 which is the result of minimizing G_3. This is shown in Figure 5.3(b). Note that the path that leads to `III` where to the write to `x` is enabled is feasible for the individual threads.

Iteration 4

Step 1_4 We recompute the reachability assuming the context has threads behaving as A_3, and the predicates Λ_2. The same abstract race as in step 1_2 is possible again.

Step 2_4 We concretize the trace from the previous step. This time, we get the feasible path `1` $\rightarrow$ `2` $\rightarrow$ `3` $\rightarrow$ `4` $\rightarrow$ `5` $\rightarrow$ `6` for the individual threads, but find that the composed trace, where the context thread follows the above path and *waits at* `6` then the main thread follows the same path to `6` is infeasible. This is because the first thread will set `state` to 1, and so the second thread cannot take the assume edge `3` $\rightarrow$ `4`. The analysis reveals the predicates $state = 0$ and $state = 1$ rule out this behavior and we add these to our set to get

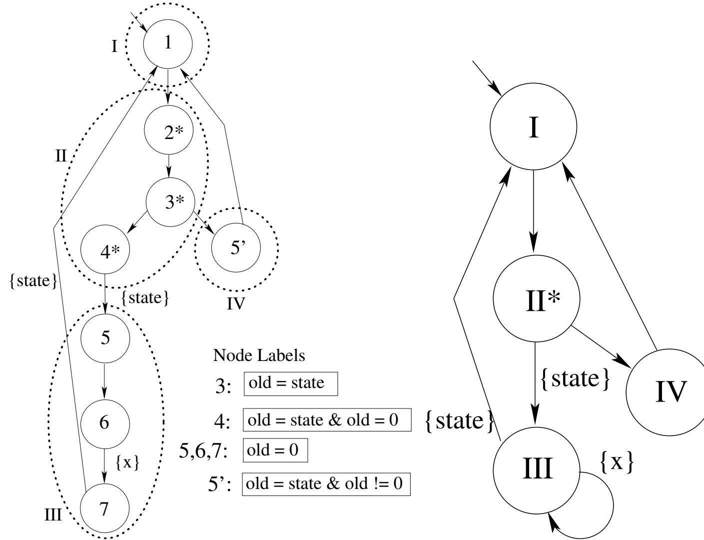

Figure 5.3: (a) ARG G_3 (b) Min. ARG A_3

Λ_4, set A_4 to be the empty AT and return to step 1.

Iteration 5

Steps $1_5, 2_5, 3_5$ We repeat the reachability using A_4 and Λ_4, to get the ARG G_5, shown in Figure 5.4. Notice that this time, the vertices in G_5 contain the values of `state`. The reach set is error free, but G_5 is not overapproximated by A_4, the latter being the empty AT, so we set A to be A_5 which is the result of minimizing G_5. This is the same as the AT shown in Figure 5.1(c). Notice that `II,III` are not collapsed as they differ on the values of predicate $state = 0$. Notice also, that in A_5, the various locations are labeled by predicates describing the value of `state` when the abstract thread is at that location. In particular, when a thread is at `IV`, the value of `state` is non-zero, thus preventing other threads from writing `x`.

Iteration 6

Step $1_6, 2_6$ We compute the ARG with the new AT A_5 with counter parameter still 1. We find a few more states, *e.g.* after a thread sees in its atomic block that `state` is 1, it may see that it has been havoced, but this is not essential as the thread still just returns to the head of the loop (since its `old` is still 0). There is no error possible as if a context AT goes first, it keeps `state` at 1 till after it has written `x`: so when the main thread takes the assume edge $\mathtt{3} \rightarrow \mathtt{4}$ ($[\mathtt{state} = \mathtt{0}]$) the region is empty ($state = 0 \wedge state = 1$ is unsatisfiable) meaning

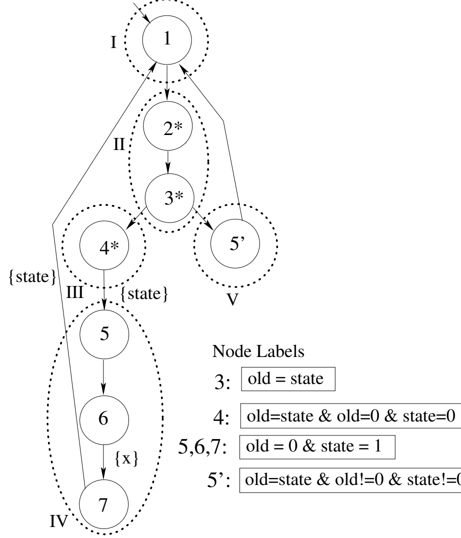

Figure 5.4: ARG G_5

that edge is not behavior is not possible. Similarly, if the main thread gets in first, when a context thread attempts to take the abstract edge $2' \to 3'$, the abstract state is empty. The resulting ARG is G_6, and we proceed to step 3.

Step 3_6 We find that in fact G_6 is overapproximated by A_5 and so the context approximation is sound. We conclude the system is free of races.

5.2 Safety Verification of Multithreaded Programs

We now formally define our model of multithreaded programs, and present our approach to verifying them. First we define the semantics of such programs abstractly using transition systems, and then one concrete syntactic representation using CFAs. Then, we shall describe our approach to verifying multithreaded programs using Thread-Context Reasoning.

For two sets of variables X, X' and an X-state s and X'-state s', we say $s \approx s'$ if for all $x \in X \cap X'$, we have $s.x = s'.x$. For sets of X, X' states S, S', we say $S \lesssim S'$ iff for each $s \in S$, there exists $s' \in S'$ such that $s \approx s'$. We say $S \approx S'$ if $S \lesssim S'$ and $S' \lesssim S$. Notice that $\approx$ is an equivalence relation and $\lesssim$ is reflexive and transitive. For an X-state s, and X'-state s', such that $s \approx s'$, we denote by $s \circ s'$ the $X \cup X'$-state that equals $s_i.x$ if $x \in X_i$ (for $i \in \{1, 2\}$). For sets of X, X'-states S, S', we write $S \circ S'$ for the set of $X \cup X'$-states $\{s \circ s' \mid s \in S, s' \in S', s \approx s'\}$.

5.2.1 Multithreaded Labeled Transition Systems

A *thread LTS* (or thread, in brief) is an LTS $T = (X, \Sigma, \rightsquigarrow, S_0)$ (cf. Section 2.1). For a label $l \in \Sigma$, the variables *written by* l are given by

$$\mathsf{Write}_T(l) = \{x \in X \mid \exists s \overset{l}{\rightsquigarrow} s'.s.x \neq s'.x\}$$

We use *Multithreaded Labeled Transition Systems* (MLTS) to model multithreaded programs. An MLTS $\hat{\mathcal{S}}$ is a set of threads $\{T_1, T_2, \ldots\}$ such that for each i, j, (1) $T_i.S_0 \approx T_j.S_0$, and, (2) $T_i.\Sigma \cap T_j.\Sigma = \emptyset$. The set of *global variables* of $\hat{\mathcal{S}}$ is $X_G = \cup_{i \neq j} T_i.X \cap T_j$. The set of *local variables* of $\hat{\mathcal{S}}$ is $X_L = X \setminus X_G$. An MLTS $\hat{\mathcal{S}}$ corresponds to a thread LTS $(X, \Sigma, \rightsquigarrow, S_0)$, defined as follows:

Variables. The set of variables X is $\cup_i T_i.X$,

Labels. The set of labels of Σ is $\cup_i T_i.\Sigma$,

Transition Relation. The transition relation $\rightsquigarrow$ is defined as $(s \circ t) \overset{l}{\rightsquigarrow} (s \circ t')$ if there exists some $T_i \in \hat{\mathcal{S}}$ such that $t \overset{l}{\rightsquigarrow}_{T_i} t'$. That is, if the thread T_i updates its variables according to its transition relation, and all the other variables remain unchanged.

Initial States. The initial states S_0 are the combination $T_1.S_0 \circ T_2.S_0 \circ \ldots$, of the initial states of all the threads.

5.2.2 Thread-Context Verification

Given an MLTS $\hat{\mathcal{S}}$, with initial states S_0, we define $\mathsf{Reach}.\hat{\mathcal{S}}$ as for LTSs (cf. Section 2.3), namely the set $\{s \mid \exists s_0 \in S_0, \sigma \in \Sigma^*.s_0 \overset{\sigma}{\rightsquigarrow} s\}$. Given a set of Y-states $\mathcal{E}$, we say $\hat{\mathcal{S}}$ is safe w.r.t. $\mathcal{E}$ if $\mathsf{Reach}.\hat{\mathcal{S}} \circ \mathcal{E} = \emptyset$. The *multithreaded safety verification problem* is to decide if an MLTS $\hat{\mathcal{S}}$ is safe w.r.t. a set of error states $\mathcal{E}$. To check if the MLTS is safe w.r.t. $\mathcal{E}$, we could attempt to compute all the states reachable from the initial states. Such an endeavour is most likely doomed to failure as the "product" of the threads will be prohibitively large, for non-trivial threads.

Assume-Guarantee Reasoning

First, we recall the technique of Assume-Guarantee [MC81; Jon83; MC81; AL91; AH99; McM97; FQS02] reasoning, in which we replace each thread with a smaller one, an assump-

tion, when verifying the system, and then, to ensure soundness, check that the assumption made captured all the behaviors of the thread it replaced. To describe this method, we formally describe what it means for one thread to capture the behaviour of another (simulation), and "all the behaviors" of a thread (projection).

Simulation. For two LTS $T_1 = (X_1, \cdot, \rightsquigarrow_1, \cdot)$ and $T_2 = (X_2, \cdot, \rightsquigarrow_2, \cdot)$, we define the *simulation relation*[HHK95; CGP99] $\preceq$ to be the largest set $\mathsf{V}.X_1 \times \mathsf{V}.X_2$ such that if $s_1 \preceq s_2$, then

1. $s_1 \approx s_2$, and,

2. For each $s_1 \rightsquigarrow_1 s_1'$, there exists a $s_2 \rightsquigarrow_2 s_2'$ such that $s_1' \preceq s_2'$.

For sets of X_1, X_2 states S_1, S_2, we say $S_1 \preceq S_2$ if for every $s_1 \in S_1$ there exists a $s_2 \in S_2$ such that $s_1 \preceq s_2$. We say that $T_1 \preceq T_2$ if $T_1.S_0 \preceq T_2.S_0$.

Projection. For an MLTS $\hat{\mathcal{S}} = \{T_1, T_2, \ldots\}$, we define the *projection of* $\hat{\mathcal{S}}$ *to* T_i, written $\hat{\mathcal{S}}{\downarrow}T_i$ to be the thread LTS $(T_i.X, T_i.\Sigma, \rightsquigarrow, T_i.S_0)$, where $s \overset{l}{\rightsquigarrow} s'$ iff there exists t, t' such that (1) $s \circ t \in \mathsf{Reach}.\hat{\mathcal{S}}$, and, (2) there exists $\sigma \in (\hat{\mathcal{S}}.\Sigma \setminus T_i.\sigma)^*$ such that $(s \circ t) \overset{l\sigma}{\rightsquigarrow}_{\hat{\mathcal{S}}} (s' \circ t')$. Intuitively, this projection is an LTS that captures the transitions made by T_i when $\hat{\mathcal{S}}$ executes from its initial states.

Theorem 9 [Assume-Guarantee] *For an MLTS $\hat{\mathcal{S}} = \{T_1, T_2, \ldots\}$, and set of states $\mathcal{E}$, if there exists $\hat{\mathcal{S}}' = \{T_1', T_2', \ldots\}$ such that for each i, letting $\hat{\mathcal{S}}_i' = \hat{\mathcal{S}}'[T_i/T_i']$, we have*

1. *$\hat{\mathcal{S}}_i'$ is safe w.r.t $\mathcal{E}$, and,*

2. *$\hat{\mathcal{S}}_i'{\downarrow}T_i \preceq T_i'$,*

then $\hat{\mathcal{S}}$ is safe w.r.t. $\mathcal{E}$.

Thread-Context Reasoning for Symmetric MLTS

The above applies to arbitrary MLTSs. We now turn to the safety verification problem for *symmetric* MLTSs, namely MLTSs comprising arbitrarily many copies of the same thread.

Let X be a set of variables partitioned into X_G, a set of global variables, and, X_L, a set of local variables. For a natural i, we write $X[i]$ be the set $X_G \cup \{x_i \mid x \in X_L\}$. For an

X-state s, we write $s[i]$ for the $X[i]$-state where $s \approx s[i]$ and for $x_i \in X_L$, $s[i].x_i = s.x$. For a set of X-states, S, we write $S[i]$ for the set $\{s[i] \mid s \in S\}$. For a thread $T = (X, \Sigma, \rightsquigarrow, S_0)$, we write $T[i]$ to be the thread $(X[i], \Sigma[i], \rightsquigarrow, S_0[i])$ where (1) $\Sigma[i]$ is the set $\{l[i] \mid l \in \Sigma\}$, and, (2) $s[i] \overset{l[i]}{\rightsquigarrow} s'[i]$ if $s \overset{l}{\rightsquigarrow} s$. For a thread T we write write T^ω for the MLTS $\{T[i] \mid i \in \mathbb{N}\}$. Notice that the global variables of the MLTS T^ω are exactly $T.X_G$. Given a thread T, and a set of error states $\mathcal{E} \subseteq$, the symmetric safety verification problem is to decide if T^ω is safe w.r.t. $\mathcal{E}$.

Given a thread T and an *environment* thread T', the *Thread-Context Program*(TCP) of (T, T'), written $\mathsf{TCP}.(T, T')$ is the MLTS $\{T\} \cup T'^\omega$. Intuitively, the TCP of T, T' is a program where a *main* thread T is executing in a *context* of an arbitrarily many copies of the *environment* thread T'. If the environment thread captures all the relevant behaviors of the main thread, then the safety of the TCP implies the safety of the symmetric program.

Proposition 7 [Thread-Context Reasoning] *For a thread T, and a set of error states $\mathcal{E}$, if there exists a T' such that:*

1. $\mathsf{TCP}.(T, T')$ *is safe w.r.t.* $\mathcal{E}$*, and,*

2. $\mathsf{TCP}.(T, T') \downarrow T \preceq T'$

then T^ω is safe w.r.t. $\mathcal{E}$.

The above follows from Theorem 9. This motivates Algorithm 5.2.2 for the symmetric safety verification problem. In Algorithm 5.2.2, we try to find an appropriate environment thread T' by starting with the "empty" thread, and then enriching its transitions until either we find a counterexample behavior, or, we find that the conditions of Proposition 7 are met, and thus T^ω is safe w.r.t. $\mathcal{E}$.

5.3 Abstractions

There are several related hurdles that must be crossed in order to implement the algorithm 5.2.2. First, we must be able to compute $\mathsf{Reach}.(\mathsf{TCP}.(T, T'))$. In order to do so, we require some abstraction of $\mathsf{TCP}.(T, T')$. Second, we must be able to compute the projection

Algorithm 4 Context Inference

Input: Thread T, Error states $\mathcal{E}$
Output: SAFE if T^ω is safe w.r.t. $\mathcal{E}$, UNSAFE τ, where τ is a counterexample to $\mathcal{E}$ otherwise

```
T' := (T.X, T.Σ, ∅, T.S_0)                                   {Initialize}
try
  repeat
    T' := TCP.(T, T')↓T                                  {Weaken assumption}
  until Reach.(TCP.(T, T')) ∘ E = ∅ and TCP.(T, T')↓T ⪯ T'
  return SAFE
with (Exception(τ)) →
  return UNSAFE(τ)
```

of $\mathsf{TCP}.(T, T')$ w.r.t. T (in order to get the new T'), as well as check if one candidate environment thread is simulated by another. In order to do these, we must have an abstraction of environment threads that can be computed by projecting the TCP, and on which we can compute the simulation.

Suppose we are given a symbolic abstraction structure that abstracts the data values for the main thread (cf. Section 2.1.2). We shall use this data abstraction structure, to obtain an abstract representation for environment threads, called a control abstraction. From the control abstraction, we shall construct an abstraction structure for contexts, using counters. Finally, we shall combine the abstractions for the main thread and the context, to get an abstraction for the TCP.

5.3.1 Main Thread: Data Abstraction

A *Havocable Region Structure* for a thread $T = (X, \Sigma, \rightsquigarrow, S_0)$, is a pair $(\mathcal{R}, \mathsf{Hvc})$ where $\mathcal{R}$ is a symbolic region structure $(R, \bot, \sqcup, \sqcap, \mathsf{post}, [\![\cdot]\!])$, for T (cf. Section 2.1.1), and $\mathsf{Hvc} : R \to 2^X \to R$, such that for all $r \in R$ and $X' \subseteq X$:

$$[\![\mathsf{Hvc}.r.X']\!] = \{s' \mid \exists s \in [\![r]\!].\forall x \in X \setminus X' s'.x = s.x\}$$

We say a region $r \in R$ is a *global* region if $r = \mathsf{Hvc}.r.X_L$, *i.e.,* all the local variables have been havoced and so can have any value. Notice that for any X', $\mathsf{Hvc}.(\mathsf{Hvc}.r.X').X' = \mathsf{Hvc}.r.X'$. A *Havocable Abstraction Structure* for a thread T is a tuple $\mathcal{A} = ((\mathcal{R}, \mathsf{Hvc}), \widehat{\mathsf{post}}, \trianglelefteq, \widehat{\mathsf{Hvc}})$, where $(\mathcal{R}, \widehat{\mathsf{post}}, \trianglelefteq)$ is a symbolic abstraction structure for T (Section 2.1.2), and $\widehat{\mathsf{Hvc}} : R \to 2^X \to R$, such that for all $r \in R$ and $X' \subseteq X$:

$$\mathsf{Hvc}.r.X' \sqsubseteq \widehat{\mathsf{Hvc}}.r.X'$$

To abstract the main thread T, we shall use a havocable abstraction structure $\mathcal{A}$ for T.

5.3.2 Environment Thread: Control Abstraction

For a symbolic region structure $\mathcal{R} = (R, \bot, \sqcup, \sqcap, \mathsf{post}, [\![\cdot]\!])$ for thread $T = (X, \Sigma, \rightsquigarrow, S_0)$, a $\mathcal{R}$-Abstract Thread is $A = (X, Q, q_0, \rightarrow)$, where:

1. X is the set of variables of T
2. Q is a finite set of abstract locations, labelled with global regions from R; we write $q\!:\!r$ for the location $q \in Q$ labeled with $r \in R$,
3. $q_0 \in Q$ is a *start location*, we write $A.r_0$ for the label of q_0,
4. $\rightarrow \subseteq (Q \times 2^X \times Q)$ is a finite set of directed edges labeled with subsets of X; an edge $(q, Y, q') \in \rightarrow$ is written $q \xrightarrow{Y} q'$.

An abstract thread A corresponds to a thread LTS T_A defined as follows:

Variables. The variables of T_A are $X \cup \{pc_A\}$, where pc_A is single local variable whose value is an element of Q, representing the abstract program counter of A,

Labels. The labels of T_A are the subsets of X,

Transition Relation. The transition relation of T_A is $\rightsquigarrow$ defined as: $s \overset{Y}{\rightsquigarrow} s'$ if there exists $q : r \xrightarrow{Y} q' : r'$ such that: (1) $s.pc_A = q$, and, $s'.pc_A = q'$, and, (2) $\{s\} \lesssim [\![r]\!]$ and $\{s'\} \lesssim [\![r' \sqcap \mathsf{Hvc}.r.Y]\!]$, and,

Initial States. The initial states of T_A are $[\![A.r_0]\!] \circ \{s \mid s.pc_A = q_0\}$.

An $\mathcal{R}$-abstract thread is *global* if all the regions labelling its locations are global, and the variables labelling its edges belong to X_G. We shall represent environment threads using global abstract threads. The global requirement is so that environment threads do not change the values of the main thread's local variables; the relevant local state of the environment threads is captured in its abstract program counter. In the sequel we shall use abstract threads in the same place as we use thread LTSs, *i.e.,* we say A for the thread LTS T_A. Now we show how to construct abstract environment threads from TCPs, and how to check if one abstract thread simulates another.

Abstract Reachability Graphs. Given a main thread T, a havocable region structure $\mathcal{R}$ for T, and an environment thread, possibly given using a $\mathcal{R}$-Abstract Thread, a $\mathcal{R}$-*Abstract Reachability Graph*(ARG) for $\mathsf{TCP}.(T, T')$ is an abstract thread $A = (X, Q, q_0, \rightarrow)$ such that there exists a function f from the states of $\mathsf{Reach.TCP}.(T, T')$ to Q with the following properties:

1. For all states $s \in \mathsf{Reach.TCP}.(T, T')$, if $f.s : r$ then $\{s\} \lesssim [\![r]\!]$,
2. For all initial states s_0 of $\mathsf{TCP}.(T, T')$, we have $f.s_0 = q_0$,
3. For each $s \in \mathsf{Reach.TCP}.(T, T')$, if $s \overset{l}{\rightsquigarrow} s'$ then (1) if $l \in T.\Sigma$, then we have $f.s \xrightarrow{Y} f.s'$ and Y contains $\mathsf{Write}_T(l)$, and, *i.e.*, all the variables written in l, and (2) if $l \notin T.\Sigma$ then $f.s = f.s'$.

An abstract reachability graph for $\mathsf{TCP}.(T, T')$ represents an overapproximation of the behavior of T in the context of arbitrarily many environment threads T'.

Proposition 8 [Abstract Reachability Graphs] *For every main thread T, havocable region structure $\mathcal{R}$ for T, and environment thread T' if A is an $\mathcal{R}$-Abstract Reachability Graph for $\mathsf{TCP}.(T, T')$, then:*

1. $\mathsf{Reach}.(\mathsf{TCP}.(T, T')) \subseteq [\![\sqcup\{r \mid q : r \in A.Q\}]\!]$,
2. $\mathsf{TCP}.(T, T') \downarrow T \preceq A$.

Proposition 8 is a generalization of Theorem 1 for reachability trees (cf. Section 3.2).

Abstract Simulation. For two $\mathcal{R}$-Abstract Threads $A = (X, Q, q_0, \rightarrow)$ and $A' = (X, Q', q'_0, \rightarrow')$, $\preceq$ is the *largest* subset of $Q \times Q'$ such that: if $q_1 : r_1 \preceq q'_1 : r'_1$ then:

1. $r_1 \sqsubseteq r'_1$, and,
2. For every $q_1 \xrightarrow{Y} q_2$ there exists a $q'_1 \xrightarrow{Y'} q'_2$ such that $Y \subseteq Y'$ and $q_2 \preceq q'_2$.

We say $A \preceq A'$ or A' *abstractly simulates* A if $q_0 \preceq q'_0$.

Proposition 9 [Abstract Simulation] $A \preceq A'$ *implies* $T_A \preceq T_{A'}$.

The above proposition, coupled with standard algorithms[CGP99; HHK95], gives us a way to check if one abstract thread simulates another.

5.3.3 Context: Counter Abstraction

Given an abstract thread, we now show how to use *counters* to construct an abstraction for *contexts* corresponding to an unbounded number of copies of the abstract thread. Intuitively, the abstract thread's location encodes all the relevant information about the *local state* of the environment thread. When constructing the abstract thread, we shall havoc all the local variables from the locations' regions and hence the havoced labels capture the relevant information about the global variables of the environment thread. Now, given that the local information corresponds directly to the abstract location, we observe that instead of tracking the local states of the each of the context threads, we can just count the number of threads at each control location[Lub84]. This still leads to an infinite number of possibilities so we shall use a *counter abstraction*, where given a parameter k we shall abstract any number greater than k to be ω (infinity).

Formally, for every $k \in \mathbb{N} \cup \{\omega\}$, we define the counter abstraction function $\alpha.k : \mathbb{N} \to \{0, \dots, k, \omega\}$ as:

$$\alpha.k.j = \begin{cases} j & \text{if } j \leq k \\ \omega & \text{otherwise.} \end{cases}$$

For an Abstract Thread A, and a natural k, the *counter abstraction* of A w.r.t. k, is the tuple $(\Delta, \delta_0, \widehat{\mathsf{post}})$ where:

1. Δ is the set of counter maps from $A.Q$ to $\mathbb{N} \cup \{\omega\}$,

2. δ_0 is the initial counter map where $\delta_0.q = \omega$ if $q = A.q_0$ and 0 otherwise, and,

3. $\widehat{\mathsf{post}}$ is a function $\Delta \to (A.Q)^2 \to \Delta$, defined as: :

$$\widehat{\mathsf{post}}.\delta.(q, q') = \begin{cases} \delta[q \mapsto \delta.q - 1][q' \mapsto \alpha.k.(\delta.q' + 1)] & \text{if } \delta.q > 0, \\ \delta & \text{o.w.} \end{cases}$$

5.3.4 Abstracting Thread-Context Programs

We now combine the above abstractions as follows. Suppose that we are given a symbolic havoc abstraction structure $\mathcal{A} = (\mathcal{R}, \widehat{\mathsf{post}}, \trianglelefteq, \widehat{\mathsf{Hvc}})$ for a thread T, a $\mathcal{R}$-Abstract Thread A, and a natural k. Let the counter abstraction of A w.r.t. k be $(\Delta_A, \delta_0, \widehat{\mathsf{post}}_A)$. We have $\mathsf{TCPabs}.(T, \mathcal{R}).(A, k)$ as the tuple $(R^\omega, [\![\cdot]\!]^\omega, \widehat{\mathsf{post}}^\omega)$, defined as follows.

Regions. $R^\omega = R \times \Delta_A$ is the set of regions of $\mathsf{TCP}.(T,A)$; *i.e.,* a region of $\mathsf{TCP}.(T,A)$ is a pair (r,δ) where r is a region for T, and δ is a countermap for A. $[\![\cdot]\!]^\omega$ is a map from R^ω to sets of $\mathsf{TCP}.(T,A)$ states, where $s \in [\![(r,\delta)]\!]^\omega$ iff there exists a map f from $\mathbb{N}$ to $A.Q$ such that (1) for all $i \in \mathbb{N}$, if $f.i : r_i$ then $\{s\} \lesssim [\![r \sqcap r_i]\!]$, and (2) for all q, $|\{i \mid f.i = q\}| \leq \delta.q$.

Abstract Post. The operations of $\mathsf{TCP}.(T,A)$ are the environment operations Y corresponding to edges $q \xrightarrow{Y} q'$ in A, and the main thread operations $l \in T.\Sigma$. An environment operation $q \xrightarrow{Y} q'$ is *enabled* in a region (r,δ), if $\delta.q > 0$, and main thread operation l in $T.\Sigma$ is always enabled. For counter map δ for the abstract thread A, we write $\sqcap\delta$ as an abbreviation for $\sqcap\{r \mid \exists q : r \text{ s.t. } \delta.q > 0\}$. The function $\widehat{\mathsf{post}}^\omega$ takes an element of R^ω, and an operation of $\mathsf{TCP}.(T,A)$, and returns another element of R^ω overapproximating the successor states corresponding to the operation. We define

$$\widehat{\mathsf{post}}^\omega.(r,\delta).\mathsf{op} = \begin{cases} (\sqcap\delta \sqcap \widehat{\mathsf{post}}.r.l, \delta) & \text{if } \mathsf{op} = l \in T.\Sigma \\ (\sqcap\delta' \sqcap \widehat{\mathsf{Hvc}}.r.Y, \delta') & \text{if } \mathsf{op} = q \xrightarrow{Y} q' \text{ and } \delta' = \widehat{\mathsf{post}}_A.\delta.(q,q') \end{cases}$$

if op is enabled in (r,δ) and $(\perp, \lambda q.0)$ otherwise.

Proposition 10 [Abstract TCP] *For every thread T, with havocable abstraction structure $\mathcal{A} = ((\mathcal{R},\cdot),\cdot,\cdot,\cdot)$, a $\mathcal{R}$-Abstract Thread A, and a $k \in \mathbb{N}$, if* $\mathsf{TCPabs}.(T,\mathcal{R}).(A,k) = (R^\omega, [\![\cdot]\!]^\omega, \widehat{\mathsf{post}}^\omega)$ *then:*

1. $\mathsf{TCP}.(T,A).S_0 \subseteq [\![r_0 \sqcap A.r_0]\!]$ *implies* $\mathsf{TCP}.(T,A).S_0 \subseteq [\![(r_0,\delta_0)]\!]^\omega$,

2. *For any $r \in R^\omega$, for every $s \in [\![r]\!]^\omega$, (1) if $s \overset{l}{\rightsquigarrow} s'$ for $l \in T.\Sigma$, then $s' \in [\![\widehat{\mathsf{post}}^\omega.r.l]\!]^\omega$, and (2) if $s \overset{l[i]}{\rightsquigarrow} s'$ for $l \in A.\Sigma$, then $s' \in [\![\widehat{\mathsf{post}}^\omega.r.l]\!]^\omega$.*

5.4 Verification by Thread-Context Abstraction-Refinement

Recall that the (symmetric) verification problem we are interested in is the following: given the (main) thread T, and a set of $T.X$ states $\mathcal{E}$, we wish to check if T^ω is safe w.r.t. $\mathcal{E}$. We now show how the abstractions described in the previous section can be used to implement Algorithm 5.2.2 for the symmetric safety verification problem. Our strategy is to *infer* an abstract thread that satisfies the conditions of Proposition 7, and thus show the multithreaded system safe.

We require a havocable abstraction structure $\mathcal{A} = ((\mathcal{R}, \cdot), \cdot, \cdot, \cdot)$, for T where $\mathcal{R}$ is the region structure $(R, \bot, \sqcup, \sqcap, \mathsf{post}, [\![\cdot]\!])$. The set $\mathcal{E}$ is given as a region in R. To implement Algorithm 5.2.2 we use:

1. $\mathcal{R}$-abstract threads to represent the candidate environment threads T',
2. $\widehat{\mathsf{post}}^{\omega}$ to compute (overapproximations of) $\mathsf{Reach}.(\mathsf{TCP}.(T, T'))$,
3. Abstract Reachability Graphs to compute $\mathsf{TCP}.(T, T') \downarrow T$ to weaken the environment assumption, and,
4. Abstract Simulation to check if the assumed environment thread simulates the behavior of T in the current context.

The Proposition 10 states that the initial abstract state of the TCP contains the initial states of the TCP, and that $\widehat{\mathsf{post}}^{\omega}$ is an overapproximation of the successor states in the Thread-Context Program. Given the operation $\widehat{\mathsf{post}}^{\omega}$, we can construct a reachability graph for a $\mathsf{TCP}.(T, T_A)$, by starting from a root node labeled with (r_0, δ_0) and constructing successors for each node by computing $\widehat{\mathsf{post}}^{\omega}$ for each enabled operation, and merging nodes with the same region. We shall use a variant of SymbReachRefine Algorithm 1, to construct this abstract reachability graph.

First, we show how if *given* an abstract thread, we can check if the system is safe, and then we show how to *infer* the abstract thread by combining the fixpoint computation of Algorithm 5.2.2, with counterexample-guided abstraction refinement.

5.4.1 Checking

Suppose that in addition to the thread T, havocable abstraction structure $\mathcal{A} = ((\mathcal{R}, \cdot), \cdot, \cdot, \cdot)$, for T and error region $\mathcal{E}$, we have a $\mathcal{R}$ -Abstract Thread A, and a natural number k. In Algorithm Check, using two steps, we directly use Proposition 7 to check that T^{ω} is safe w.r.t. $[\![\mathcal{E}]\!]$.

1. Assume that A is a sound approximation of the behavior of T when T is composed with infinitely many copies of itself. Compute the set of abstract states reachable in the abstract multithreaded program $\mathsf{TCP}.(T, A)$, using the abstraction $\mathsf{TCPabs}.(T, \mathcal{R}).(A, k)$

Algorithm 5 $\mathsf{ReachAndBuild}.(\mathcal{A}, r_0, \mathcal{E}).(A, k)$

Input: A havoc abstraction structure $\mathcal{A} = ((\mathcal{R}, \cdot), \cdot, \cdot, \cdot)$ for thread LTS T, initial region $r_0 \in \mathcal{R}.R$, error region $\mathcal{R}.R$, a global $\mathcal{R}$-Abstract Thread A, natural k.

Output: Either a $\mathcal{R}$-Abstract Thread A', s.t. $\mathsf{TCP}.(T, A) \downarrow T \preceq A'$, or raise exception $\mathsf{Exception}(\tau)$, where τ is a sequence of $\mathsf{TCP}.(T, A)$ operations.

1: $(R^\omega, \cdot, \widehat{\mathsf{post}}^\omega) := \mathsf{TCPabs}.(T, \mathcal{R}).(A, k)$
2: $L := \{(r_0, \delta_0)\}$, $\mathit{Intl} := \emptyset$, $G := (\emptyset, \emptyset, \emptyset)$
3: **while** $L \neq \emptyset$ **do**
4: pick and remove state (r, δ) from L
5: **if** not $((r, \delta) \in \mathit{Intl})$ **then**
6: $\mathit{Intl} := \mathit{Intl} \cup \{(r, \delta)\}$
7: **if** $r \sqcap \mathcal{E} \not\sqsubseteq \bot$ **then**
8: $\tau := \mathsf{FindPath}.\mathit{Intl}.(r_0, \delta_0).(r, \delta)$
9: **raise** $\mathsf{Exception}(\tau)$
10: **else**
11: **for each** enabled operation op of $\mathsf{TCP}.(T, A)$ **do**
12: $(r', \delta') := \mathsf{post}.(r, \delta).\mathsf{op}$
13: $\mathsf{Connect}.\mathcal{R}.G.(r, \mathsf{op}, r')$
14: $L := L \cup \{(r', \delta')\}$
15: $A' := (T.X, G.Q, \mathsf{Find}.G.r_0, G.\xrightarrow{\ \rangle\ }$
16: **return** A'

and check that this set does not contain any error states. If an error is reached, return "possibly UNSAFE." This step is implemented by procedure ReachAndBuild, which does a reachability analysis and also builds an abstract reachability graph G describing the behavior of T when its context is an arbitrary number of abstract threads A running concurrently.

2. Guarantee that the abstract thread A is indeed a sound approximation of the behavior of T in this context, by checking that abstract reachability graph G computed in the previous step is overapproximated by A, or more precisely, that $G \preceq A$. If the check succeeds, return SAFE, else return "possibly UNSAFE." We perform this check by using a variation of the standard simulation checking algorithm [HHK95].

The soundness of the above follows via inductive "assume-guarantee" reasoning [Jon83; AH99]. We now describe ReachAndBuild in greater detail.

Procedure ReachAndBuild is shown in Algorithm 5. It is a standard worklist based reachability algorithm [CGP99], but additionally builds an abstract reachability graph G summarizing the reachability information similar to how Algorithms 1,2 build reachability

Algorithm 6 $\mathsf{Connect}.\mathcal{R}.G.(r, \mathsf{op}, r')$

Input: Region Structure $\mathcal{R}$ for thread T, Augmented Graph $G = (Q, \rightarrow, S)$, where $S : Q \rightarrow 2^{\mathcal{R}.R}$, and (r, op, r') where $r, r' \in R$ and op is in $T.\Sigma \cup 2^{T.X}$.

$q := \mathsf{Find}.G.r$; $q' := \mathsf{Find}.G.r'$

if $\mathsf{op} \in T.\Sigma$ **then**

{Main thread operation}

if $q \xrightarrow{Y} q'$ is in G for some Y **then**

Replace $(n \xrightarrow{Y} n')$ with $(n \xrightarrow{Y \cup \mathsf{Write}_T(\mathsf{op})} n'$ in G

else

Add $q \xrightarrow{\mathsf{Write}_T(\mathsf{op})} q'$ to G

else

$\mathsf{Union}.G.(q, q')$ {Environment operation}

Algorithm 7 $\mathsf{Find}.G.r$

Input: Aug. Abstract Thread $G = (Q, \rightarrow, S)$, where $S : Q \rightarrow 2^R$, and $r \in R$.

Output: A location $q \in Q$

if $\exists q \in Q : r \in S.q$ **then**

return q

else

Create a fresh location $q\!:\!r$

$Q := Q \cup \{q\!:\!r\}$

$S := S[q \mapsto \{r\}]$

return q

---

trees. The main loop of lines 3–14 runs the reachability construction, using the worklist L. At each step, a state is chosen from the worklist. If it has not been seen before (line 5), it is added to the set of explored states (line 6), and checked for possible errors. If an error state has been hit (line 7), the procedure finds an (abstract) interleaved error trace to the error state, and raises an exception containing the error trace. Otherwise, the current state is expanded. For this, we construct the successor of the current state for each operation enabled from it (line 12), and connect the current state and its successor as an edge in the abstract reachability graph G, using the procedure Connect described shortly. Finally, the successor states are added to the worklist. Note that the locations of the abstract reachability graph G correspond to abstract thread states, as we drop the counter maps corresponding to the context's abstract state. At the end of the while loop, we return A' which is the abstract thread corresponding to G; the initial location is the location corresponding to the initial region r_0.

Algorithm 8 Algorithm Union

Require: Region Structure $\mathcal{R} = (R, \cdot, \sqcup, \cdot, \cdot, \cdot)$, Aug. Abstract Thread $G = (Q, \rightarrow, S)$ where $S : Q \rightarrow 2^R$, and $q, q' \in Q$.

  if $(q \neq q')$ **then**
    Let $q\!:\!r$, and $q'\!:\!r'$
    Relabel q with $r \sqcup r'$
    $S := S[q \mapsto S.q \cup q']$
    for each $q'' \xrightarrow{Y} q'$ in G **do**
      Add $q'' \xrightarrow{Y} q$ to G
      Remove $q'' \xrightarrow{Y} q'$ from G
    for each $q' \xrightarrow{Y} q''$ **do**
      Add $q \xrightarrow{Y} q''$ to G
      Remove $q' \xrightarrow{Y} q''$ from G
  return q

Procedure $\mathsf{Connect}$ adds edges between the abstract states computed by the reachability analysis (Algorithm 6). It takes as argument the augmented Abstract Thread G that is being constructed, abstract thread states r and r' (the successor of r), and an operation op. Each location of G corresponds to a set of thread regions. The Abstract Thread G is augmented with a map S that maps a location q to the set $S.q$ of thread regions mapped to q. $\mathsf{Connect}$ first finds locations q, q' corresponding to r and r' respectively by invoking the procedure Find. When Find is called with abstract thread state r it checks if a there exists a location q with $r \in S.q$; If so, it returns that location and if not, it returns a new location q where $S.q = \{r\}$. An invariant maintained is that if $q : r$ then $r = \sqcup S.q$. The edges of the graph G are added depending on the type of the operation op. There are two cases: op is either a main thread operation, or an environment operation. In the first case, we add the edge $q \xrightarrow{\mathsf{Write}_T(\mathsf{op})} q'$, labelled with the variables written by the main thread, if an edge exists between q, q', we add the written variables to the edge label. In the second case,*i.e.,* if the operation is an environment operation, then the two locations q and q' are unified by procedure Union which "merges" q' with q by merging the regions labelling q, q', adding all the in and out edges, and deleting q' and all its edges from G.

Proposition 11 [Soundness of $\mathsf{ReachAndBuild}$**]** *For any havoc abstraction structure $\mathcal{A}$ containing the havoc region structure $\mathcal{R}$, for thread T, region r_0, and $\mathcal{R}$-Abstract Thread A, such that $T.S_0 \subseteq [\![r_0 \sqcap A.r_0]\!]$ if $\mathsf{ReachAndBuild}.(\mathcal{A}, r_0, \mathcal{E}).(A, k)$ returns A', then:*

1. $\mathsf{TCP}.(T, A)$ *is safe w.r.t. $\mathcal{E}$, and,*

2. $\mathsf{TCP}.(T,A)\downarrow T \preceq A'$.

Using Proposition 10, we can show that A' is an Abstract Reachability Graph for $\mathsf{TCP}.(T,A)$, and hence the two facts above follow from Proposition 8 about Abstract Reachability Graphs.

If in addition the Abstract Thread A' (returned by $\mathsf{ReachAndBuild}$), is abstractly simulated by A, then by Proposition 9, we have $A' \preceq A$. This, combined with the second fact from Proposition 11 and transitivity of $\preceq$ implies that $\mathsf{TCP}.(T,A)\downarrow T \preceq A$. Hence, A is an environment thread that meets the requirements of Thread-Context Reasoning (Proposition 7), and so we can conclude that T^ω is safe w.r.t. $\mathcal{E}$.

5.4.2 Inference

Algorithm 9 $\mathsf{CIRC}.(\mathcal{A},\mathsf{Refine},r_0,\mathcal{E})$

Input: A havoc abstraction structure $\mathcal{A}$ containing the region structure $\mathcal{R}$ for thread LTS T, initial region $r_0 \in \mathcal{R}.R$ s.t. $T.S_0 = [\![r_0]\!]$, error region $\mathcal{R}.R$, and a refine operator Refine.

Output: SAFE if $\mathsf{TCP}.(T,)$ is safe w.r.t. $[\![\mathcal{E}]\!]$, UNSAFE σ where σ is a T^ω counterexample to $[\![\mathcal{E}]\!]$.

1: $k := 0$
2: **while** *true* **do**
3:   **try**
4:     $A' = \emptyset$
5:     **repeat**
6:       {Inner loop: build environment thread}
7:       $(A,\mu) := \mathsf{Collapse}.\mathcal{A}A'$
8:       $A' := \mathsf{ReachAndBuild}.(\mathcal{A},r_0,\mathcal{E}).(A,k)$
9:     **until** $(A' \preceq A)$
10:     **return** SAFE
11:   **with** $(\mathsf{Exception}(\tau)) \rightarrow$
12:     **if** $\mathsf{Refine}.T.A.A'.\mu.\tau = \text{REAL }(\sigma)$ **then**
13:       **return** UNSAFE (σ)
14:     **else**
15:       $(r_0', k') := \mathsf{Refine}.T.A.A'.\mu.\tau$ $\{r_0' \trianglelefteq r_0 \text{ or } k < k'\}$
16:       $r_0 := r_0'$
17:       $k := k'$ {Outer loop: refine abstraction}
18: **done**

In general, the abstract environment thread A that succinctly summarizes the behavior of thread T, and is simultaneously precise enough to show the absence of concurrency errors, is not available. Therefore, we must construct this abstraction automatically via an

inference algorithm. Algorithm 5.4.2 shows our inference algorithm CIRC.

The algorithm has an outer loop within which the abstraction is successively refined when required, and an inner loop, which attempts to build an appropriate context A. In the first iteration of the outer loop, the initial abstraction comes from r_0 and the counter abstraction parameter k is 0. In the first iteration of the innerloop we assume the that each environment thread does nothing, *i.e.,* A' is set to the empty abstract thread (line 4). We then minimize the abstract reachability graph A' with the procedure Collapse (line 6). Collapse takes abstract reachability graph A' and returns its weak bisimulation quotient Abstract Thread A [CGP99], together with a map μ that maps each location of A' to its equivalence class (location) in A. In the first iteration, this is still empty. At each iteration, A will be the "current" approximation of the main thread behaving in the context of an unbounded number of copies of itself. We then call ReachAndBuild to see how T behaves in the context of an unbounded number of copies of A, the result being the new abstract thread A' (line 7). If the present approximation A simulates the new abstract thread A' then it means that A was a sound approximation (*i.e.,* meets the "guarantee"), and we break out of the loop and return SAFE (lines 8, 9). If on the other hand, we find that A was not a good approximation (fails to meet the guarantee) then we repeat the loop with the new abstract thread A', which now gives us a better approximation of how each context thread behaves (the **repeat**...**until** loop of lines 5–8).

At any point, the procedure ReachAndBuild may raise an exception claiming it has an abstract error trace to $\mathcal{E}$. We trap this exception and analyze the counterexample to see if it is genuine, and if not, obtain a more precise thread region r'_0 or a larger (and hence, more precise) counter parameter (lines 10–16). The exception Exception(τ) is caught in line 10, and checked in procedure Refine. Procedure Refine takes as input a T, an Abstract Thread A, the abstract reachability graph A' such that A is the bisimilarity quotient of G, the map μ mapping states of A' to those of A, and an abstract error trace τ, which is a sequence of operations that are either the main thread's operations or environment operations (corresponding to edges in A). If τ can be realized in the concrete program, Refine returns a real error REAL together with a concrete interleaved trace σ, which is a counterexample to the error states $[\![\mathcal{E}]\!]$ of T^ω. In this case, the algorithm Refine returns

UNSAFEtogether with the real error σ. On the other hand, if the current error path τ does not have a concrete realization in T^ω then Refine, a variant of the procedure described in Section 3.3, returns a refinement of the abstraction by returning either a new, more precise initial region r_0, or a larger value of the counter, k'. The algorithm updates the thread and context abstractions by using the more precise root, and the larger counter respectively (lines 14–16). If the abstraction is updated, we repeat the outer loop by resetting the current approximation of each context thread A' back to the empty abstract thread, and repeat the inner loop using the new abstraction given by r_0, k.

Procedure Collapse The procedure Collapse takes a havoc abstraction structure $\mathcal{A}$, containing the havoc region structure $(\mathcal{R}, \mathsf{Hvc})$, for thread T, and a $\mathcal{R}$-abstract thread G and returns (1) a *global* $\mathcal{R}$-abstract thread A which is the a weak bisimulation quotient G after replacing the location labels with their global versions, and, (2) a mapping μ from $G.Q$ to $A.Q$ mapping each location of G to its equivalence class (location) in A. The algorithm Collapse works in two steps. First, for each location $q : r \in G.Q$, we relabel the location with $\mathsf{Hvc}.r.X_L$, where X are the variables of T Also, we remove all local variables X_L from the sets labeling the edges of G. We then run a standard weak bisimilarity algorithm [CGP99] with the resulting global regions labeling states of G as observables. The bisimilarity procedure also constructs the required mapping. Whenever in G we have $q \xrightarrow{Y} q'$, and the bisimilarity collapses q, q' to the same location q'' in A, we ensure that A has a self loop edge $q'' \xrightarrow{Y'} q''$ with $Y \subseteq Y'$. The result is a global $\mathcal{R}$-Abstract Thread. This is important as we want that any local variable appearing in the analysis refers to the local of the main thread. The global abstract thread A returned when Collapse.$\mathcal{R}.G$ is such that $G \preceq A$. Putting the above together, we get the correctness of CIRC.

Theorem 10 [Correctness of CIRC] *If Algorithm* CIRC *on input $\mathcal{A}$ containing a region structure $\mathcal{R}$ for a thread LTS T, a refine operator* Refine*, initial region $r_0 \in \mathcal{R}.R$, and error region $\mathcal{E} \in \mathcal{R}.R$, terminates and returns:*

1. SAFE *then T^ω is safe w.r.t $[\![\mathcal{E}]\!]$.*

2. UNSAFE*(σ) then T^ω is not safe w.r.t. $[\![\mathcal{E}]\!]$.*

The algorithm is not complete in general, since reachability is undecidable already for single-threaded programs. For finite-state systems and just predicate and control state abstractions, completeness follows from finiteness of the state space. It is not obvious that even for finite-state threads, Algorithm CIRC is complete, since we consider unboundedly many threads. We show in Section 5.7 that our method using counters is complete if the threads are finite-state. This implies that the only way the procedure can loop forever is if either the threads are recursive, or we keep finding increasingly precise r_0, *e.g.* by discovering new predicates.

5.5 Race Detection for Multithreaded Imperative Programs

One instance of the previous algorithm is analyzing multithreaded imperative programs, *e.g.* those written in C. In particular, we have applied the above to look for, and guarantee the absence of data races in multithreaded C programs.

We now describe how to model multithreaded imperative programs as MLTSs, describe the race detection problem as a safety verification problem for MLTSs, describe the abstractions we use and the Refine operator for such programs. For clarity we describe our method only for CFAs for PI programs; the described method is extended in the implementation, to deal with programs with pointers, in the same way as for sequential programs.

5.5.1 MLTSs from Imperative Programs

A *Thread Control Flow Automaton*(TCFA), is a tuple $C = (X, PC, pc_0, \to)$ where $(X, PC, pc_0, \to)$ is a CFA (defined in Section 2.2.1), and X is partitioned into X_G and X_L, respectively the sets of local and global variables.

A *Thread Program* P is pair $(F, f_{\texttt{main}})$, where F is a set of functions, and each function $f \in F$ is denoted by its TCFA C_f, and $f_{\texttt{main}}$ is a special initial function in F. A thread program P corresponds to a thread LTS $T_P = (X, \Sigma, \to, S_0)$, where the first three elements are exactly the LTS corresponding to P (defined in Section 2.2.1). The global variables of P are $\cup_{f\in P.F} C_f.X_G$, and the remaining variables, as well as the program counter and call stack are locals.

A *Multithreaded Program* $\hat{P}$ is a set of thread programs $\{P_1, P_2, \ldots\}$, such that for any

pair P_i, P_j, where $i \neq j$, the locals of P_i and P_j are disjoint. The Multithreaded Program $\hat{P}$ corresponds to an MLTS $\hat{\mathcal{S}}_{\hat{P}} = \{T_{P_1}, T_{P_2}, \dots\}$. The variables of $\hat{P}$ are $\cup_i P_i.X$. The global variables of $\hat{P}$ are $\cap_{i \neq j} P_i.X \cap P_j.X$. We consider *symmetric multithreaded programs*, which arise from an arbitrary number of copies of the same thread program running concurrently.

5.5.2 Predicate Abstraction

For threads given as imperative programs, we shall use a havocable abstraction structure based on predicate abstraction (Section 2.2.2). It suffices only to describe Hvc and $\widehat{\mathsf{Hvc}}$, for data regions (Section 2.2.2); for the program counter or stack we simply replace them with $\perp$, indicating they may have any value. For a formula φ, and set of variables $X' = \{x_1, \dots, x_k\}$,

$$\mathsf{Hvc}.\varphi.X' = \exists x'_1 \dots x'_k.\varphi[x'_1, \dots, x'_k / x_1, \dots, x_k]$$

We do away with the quantifier by skolemizing, *i.e.,* substituting each x_i with a brand new x'_i. We get $\widehat{\mathsf{Hvc}}.\varphi.X'$ by replacing all atomic predicates in φ that contain a variable in X' with unknown,*i.e.,* replacing them with *true* (*false*) if the atomic predicate appears positively (negatively).

5.5.3 The Race Detection Problem

A specific instance of the safety verification problem for MLTSs is the *race detection problem.*

For a variable $x \in T.X$, define

$$\mathsf{Wr}.T.x = \{s \mid \exists s \overset{l}{\leadsto} s'.x \in \mathsf{Write}_T(l)\}$$

Suppose that we are supplied a map $\mathsf{Rd}.T$ from $T.X$ to sets of $T.X$-states. Wr (resp. Rd) stipulate the states in which the thread T writes (resp. reads) the variable x.

For LTS corresponding to imperative programs, each l is a pair $(\mathsf{op} : pc)$ (cf. Section 2.2.1). Hence $\mathsf{Write}_{T_P}((\mathsf{op}:\cdot))$ contains x if op is an assignment to x in P, and $\mathsf{Wr}.T_P.x$ is the set of all states where there is an outgoing CFA edge labeled by an assignment to x. Similarly $\mathsf{Rd}.T_P.x$ is the set of all states where there is an outgoing CFA edge labeled by an assignment $\mathtt{y} := e$ and x is a variable of e, or an assume $\mathtt{assume}[\mathtt{p}]$ and x is a variable of p.

For LTS corresponding to Abstract Threads, each l corresponds to a set of variables Y which equals $\mathsf{Write}_{T_A}(l)$. Hence, $\mathsf{Wr}.T_A.x$ is the set of all states, where the AT is at a

location with an outgoing edge labeled by a set Y that contains x. On the other hand, $\mathsf{Rd}.T_A.x = \emptyset$.

For an MLTS $\hat{\mathcal{S}} = \{T_1, T_2, \ldots\}$, and a global variable $x \in \hat{\mathcal{S}}.X_G$, we define the *race states* $\mathcal{E}_x$ to be the set of $\hat{\mathcal{S}}.X$-states:

$$\cup_{i \neq j} \mathsf{Wr}.T_i.x \cap (\mathsf{Wr}.T_j.x \cup \mathsf{Rd}.T_j.x)$$

The *race-detection problem* for a multithreaded program $\hat{P}$ and a global variable x is to check $\mathsf{Reach}.\hat{P} \circ \mathcal{E}_x = \emptyset$. We say a program P has no races on variable x iff the program P is safe w.r.t. $\mathcal{E}_x$. If $\hat{P} = P^\omega$ for some single thread program P, then the problem is called the symmetric race detection problem.

Memory Model So far we have described our algorithms without considering pointers. In our implementation, we extend the basic algorithm to deal with pointer variables and aliasing. The problem is that we cannot infer the global memory address being accessed syntactically by looking at the name of the lvalue. Thus, for the error check, we ask for every pair of lvalues l_1, l_2 at a state, if the addresses of l_1 and l_2 can be the same, and in addition if there is a race between l_1 and l_2. As an optimization, we use a flow insensitive alias and escape analysis to curtail the possible aliasing relationships to be explored.

5.5.4 Procedure Refine

The procedure Refine analyzes abstract counterexamples, to either extract genuine error traces or to refine the abstraction to eliminate the false positive. An abstract trace is a sequence of operations of the main thread and the context ACFA. The input to Refine is a CFA C, an ACFA A, a reachability graph G such that A is the weak bisimilarity quotient of G, a map μ from states of G to states of A, and an abstract trace τ. It returns either a real error trace $\bar{s}$ or, if τ is infeasible, a refinement of the abstraction (P', k') where P' is a set of predicates, and k' is a new counter value. Procedure Refine works in two steps:

1. Computing an Interleaving. A scan over the entire trace suffices to check if the parameter k is large enough. If not, the only refinement is that k is incremented. If k is large enough, we compute the *number* of context threads that participate in the counterexample. Each operation in the abstract trace is a either a main thread operation, or an abstract

T1: `I` $\rightarrow$ `II`	`skip`	*true*
T1: `II` $\rightarrow$ `III`	`old := state`	$\langle old, 1\rangle = \langle state, 1\rangle$
	`assume[state = 0]`	$\langle state, 1\rangle = 0$
	`state := 1`	$\langle state, 2\rangle = 1$
	`assume[old = 0]`	$\langle old, 1\rangle = 0$
T0: `skip`	`skip`	*true*
T0: `old := state`	`old := state`	$\langle old, 2\rangle = \langle state, 2\rangle$
T0: `assume[state = 0]`	`assume [state = 0]`	$\langle state, 2\rangle = 0$
T0: `state := 1`	`state := 1`	$\langle state, 3\rangle = 1$
T0: `assume` $old = 0$	`assume [old = 0]`	$\langle old, 2\rangle = 0$

Figure 5.5: Abstract trace, concrete interleaving, TF

operation by a specific abstract context thread. To generate the concrete interleaving, we get a concrete sequence of thread operations from the abstract context operation, by using the underlying reachability tree of the ACFA.

2. Analyzing an Interleaving. Given an interleaved trace, we must check if it is feasible. We first compute a *trace formula* (TF) which is a version of the strongest postcondition of the trace. Each operation of the trace yields a clause and the TF is the conjunction of all the clauses. The trace is feasible, and hence, the counterexample genuine, iff the TF is satisfiable, which can be checked by querying a decision procedure. If it is not satisfiable, the *proof of unsatisfiability* of the TF can be mined for predicates using an extension of the technique described in Section 3.3.

Example 14 [Refinement] In Figure 5.5 the left, middle and right columns show respectively, the abstract trace, concrete trace, and the unsatisfiable TF for the error trace from iteration 4 of the example from Section 2. The proof of unsatisfiability yields the predicates $state = 0$ and $state = 1$. □

5.6 Experiences

nesC [GLvB+03] is a programming language for networked embedded systems. It is used to implement event driven applications in the TinyOS operating system [HSW+00]. TinyOS has two sources of concurrency: *tasks* and *events*. When an interrupt occurs an event is fired, which may in turn fire other events. As other interrupts can occur while this is happening, events can preempt each other. Events may also post tasks, which are run when

Name	Variable	♯ Preds	AT size	Time
secureTosBase (9539 lines)	gTxState	11	23	7m38s
	gTxByteCnt	4	13	1m41
	gTxRunningCRC	4	13	1m50s
	gTxProto	0	9	12s
	gRxHeadIndex	8	64	20m50s
	gRxTailIndex	0	5	2s
surge (9697 lines)	rec_ptr	4	23	1m18s
	gTxByteCnt	4	15	1m34
	gTxRunningCRC	4	15	1m45s
	gTxState	11	35	9m54s
sense (3019 lines)	tosPort	6	26	16m25s

Table 5.1: Experimental results with CIRC on a 2GHz IBM T30 with 512M RAM

nothing else is happening. A task may be preempted by events, but is never preempted by another task. The presence of concurrent execution leads to potential data races on the shared state. Since tasks are nonpreemptible, there is no data race on variables accessed only in tasks, but there may be be races between events and tasks, or between two events.

As it is essential to avoid data races, nesC provides *atomic sections* in the language with an `atomic` keyword. Code in an atomic section is executed atomically. Notice that this can be modeled in our formalism as a special shared mutex variable. The nesC compiler implements a flow based static analysis to catch race conditions on shared data variables. It runs an alias analysis to detect which global variables are accessed (transitively) by interrupt handlers, and then checks that each such access occurs within an `atomic` section. However, this analysis precludes the use of some common programming idioms (*e.g.* the example from Section 2) which cause the analysis to return false positives. For this, nesC provides a `norace` annotation that the programmer must provide if she believes that there is no race condition on a data variable. In practice, almost all shared accesses are put in atomic sections to prevent compiler warnings, even though there may be no actual race condition. Since atomic sections are implemented by interrupt disabling, this may make the system less responsive. Thus, nesC programs gave us a suitable application for a precise race checker like CIRC: first, they critically require the absence of data races, and second, they use several non-trivial synchronization idioms.

Running CIRC on nesC Programs. We focused on the variables that had raised false alarms with the flow-based analysis, and which subsequently were flagged with the `norace`

qualifier. nesC programs are compiled into C and event fires translate to function calls. We modeled the nesC applications as an arbitrary number of threads each executing a big while-loop that triggered hardware interrupts non-deterministically (as long as interrupts were enabled, modeled by adding a special global) or called tasks non-deterministically (as long as nothing else was running).

Our results on some of the largest nesC applications are summarized in (Table 1). The examples requiring no predicates are ones that were trivially safe as they were accessed in atomic sections or only by tasks and our tool finds this quickly. "Line" is the number of lines in the compiled C source code. "Preds" is the number of predicates discovered to prove safety, "AT" is the number of states in the final AT. The counter parameter was always 1.

State Variable based Synchronization. Many of the variables `gTxByteCnt, gTxRunningCRC` were protected by a state variable much like the example in Section 2, and CIRC is able to show there are no races, by finding the appropriate abstraction. `gTxState` is protected in a similar manner but is accessed in a more complicated pattern: CIRC first reported a violation on it in secureTosBase. On inspection we found that the variable was accessed at several places inside a function, in most places *before* a call that changed the state variable, but at the point of conflict, it was accessed *after* changing the state variable. On moving the access to before the call, CIRC reported the system was safe. There was another "unprotected" access, that occurred when a certain function call returned the value 0, but CIRC verified that in that context, the function never returned 0. `gRxHeadIndex` uses a complicated synchronization on multiple values of a state variable along with "conditional" accesses.

Split-phase based Synchronization. The variable `rec_ptr` in surge was accessed by an interrupt handler (event) (I) and by a task (T) in the following manner: the handler fired only when I was enabled. It then disabled the interrupt I, posted the task T and then wrote to `rec_ptr`. The task, when it got to run, wrote to the variable, and then re-enabled the interrupt. This is an instance of a *split-phase* operation, used to break up long tasks. When we modeled this interrupt precisely by tracking its status in a global flag, CIRC was able to report the absence of races after inferring the appropriate AT. (Since the C code does not match up interrupt bits with handlers, we had to refer to the underlying hardware model.)

A more complicated form of synchronization was in sense where the variable `tosPort` was protected by a combination of this and a state variable. We discovered this as CIRC found a race where an interrupt fired which reset the state after one thread had already set it and was about to write to `tosPort` thus letting another thread come in and access `tosPort`. The programmer pointed out that the malicious middle interrupt was only enabled after the first thread had finished writing to `tosPort`. On modeling this interrupt, the tool was able to prove safety.

5.7 Completeness of Counter Abstractions

For finite-state threads, counterexample-guided refinement using counter abstractions terminates.

Let $T = (X \cup \{pc\}, \Sigma, \rightsquigarrow, \{s_0\})$ be a thread with the single local variable pc. As before, let T^ω be the symmetric MLTS running an unbounded number of copies of the thread T, each of which is obtained by renaming pc to $pc[i]$, which is the only local variable of thread $T[i]$. A thread T is *finite-state* if each variable in $X \cup \{pc\}$ takes values over a finite domain. The parameterized multithreaded program T^ω is finite-state if T is finite-state. The assumption that there is a single local variable is w.l.o.g. as several locals can be modeled with a single local which takes on a richer set of values. For a finite state thread T, with variables $X \cup \{pc\}$, and a set $\mathcal{E}$ of X-states, specifying possible error states, recall that the *symmetric* safety verification problem is to check if $\mathsf{Reach}.(T^\omega) \circ \mathcal{E}$. If so, we say that T^ω is *safe w.r.t.* $\mathcal{E}$, otherwise, T^ω is unsafe.

A state of T^ω can be represented by a $X \cup \{\delta\}$-state s where $\delta : Q \rightarrow \mathbb{N} \cup \{\omega\}$, is a map where Q is the finite range of the variable pc. Intuitively, $s.x$ is the value of the global variable x, and, $s.\delta.q$ for $q \in Q$ counts the number (possibly infinite) of threads $T[i]$ with $pc[i] = q$.

We now define *counter abstractions* (T, k) of the multithreaded program T^ω. Given $T = (X \cup \{pc\}, \Sigma, \rightsquigarrow, S_0)$, let the *counter abstraction* of T^ω, w.r.t. k, be the MLTS $(T, k) = ((X \cup \{\delta\}, (Q \times \Sigma \times Q), \rightsquigarrow^k, \{s_0^k\}))$, defined as follows.

Variables. The variables of (T, k) are $X \cup \{\delta\}$ where δ is a variable, not in X, that is a map from Q to $\mathbb{N} \cup \{\omega\}$.

Labels. The labels of (T, k) are triples in $Q \times \Sigma \times Q$.

Transition Relation. The transition relation of (T, k) are triples $(s, (q, l, q'), s') \in \rightsquigarrow^k$ s.t.

1. $s.\delta.q > 0$
2. there exist $t[pc \mapsto q] \overset{l}{\rightsquigarrow} t'[pc \mapsto q']$ s.t. $s \approx t$, $s' \approx t'$, and,
3. $s'.\delta = s.\delta[q \mapsto s.\delta.q - 1][q' \mapsto \alpha.k.(s.\delta.q' + 1)]$

Recall that $\alpha.k.j = j$ if $j \leq k$ and ω otherwise.

Initial States. The initial states of (T, k) are $\{s_0\}$, where s_0 is an $X \cup \{\delta\}$-state such that: $s_0 \approx T.s_0$ and $s_0.\delta = (\lambda q.$ if $q = T.s_0.pc$ then ω else $0)$.

For every k, the MLTS (T, k) overapproximates the behavior of T^ω.

Proposition 12 [Counter Abstractions] *For every T and for every $k \in \mathbb{N}$ we have:*

1. $T^\omega \overset{\preceq}{\succeq} (T, \omega) \preceq (T, k+1) \preceq (T, k)$
2. $\mathsf{Reach}.(T^\omega) \approx \mathsf{Reach}.(T, \omega) \lesssim \mathsf{Reach}.(T, k+1) \lesssim \mathsf{Reach}.(T, k)$

The second fact follows from the first using the observation that $T \preceq T'$ implies that $\mathsf{Reach}.T \lesssim \mathsf{Reach}.T'$. The above Proposition 12 states that the set of reachable states of T^ω and (T, ω) are equivalent, hence, to check if $\mathsf{Reach}.(T^\omega)$ is safe w.r.t. $\mathcal{E}$, it suffices to check if (T, ω) is safe w.r.t. $\mathcal{E}$. Additionally, it states that for k, the reachable states of (T, k) overapproximate the reachable states of (T, ω) (and hence T^ω), and increasing k results in a more precise overapproximation.

For a trace σ, we say that σ is a *leads to $\mathcal{E}$ from S_0 for P* if there exists $s_0 \in S_0$ and s such that $s_0 \overset{\sigma}{\rightsquigarrow}_P s$ and $\{s\} \lesssim \mathcal{E}$. We say σ is a counterexample to $\mathcal{E}$ for P if σ leads to $\mathcal{E}$ from $P.S_0$. For a trace $\sigma \in (Q \times \Sigma \times Q)^*$, and a sequence $\bar{i} = i_1, \ldots, i_{|\sigma|}$, we write $\sigma \circ \bar{i}$ for the trace $\sigma.1[i_1], \ldots, \sigma.|\sigma|[i_{|\sigma|}]$. From the first fact in Proposition 12 it follows that if σ is a counterexample to $\mathcal{E}$ from for (T, ω), then there exists an $\bar{i}$ such that $\sigma \circ \bar{i}$ is an counterexample to $\mathcal{E}$ for T^ω, *i.e.,* every counterexample for (T, ω) can be appropriately relabeled to get a counterexample for T^ω.

The above proposition indicates that to solve the safety verification problem for T^ω it suffices to solve it for (T, ω); if the latter is safe, then so is the former, and if the latter is unsafe, then any counterexample for it corresponds to a counterexample for the former. To analyze the latter, we shall use counter abstractions, *i.e.,* instead of analyzing (T, ω) which is infinite state, we shall analyze (T, k) which is finite, and, as the next theorem shows, sufficiently precise due to the existence of an appropriate k.

Theorem 11 [Completeness] *Let $T = (X \cup \{pc\}, \rightsquigarrow, \{s_0\})$, be a finite-state thread. Let $\mathcal{E}$ be a set of X-states.*

1. *If (T, ω) is safe w.r.t. $\mathcal{E}$, then there exists a k s.t. (T, k) is safe w.r.t. $\mathcal{E}$.*
2. *If (T, ω) is unsafe w.r.t. $\mathcal{E}$, then there exists a k and a σ that is counterexample to $\mathcal{E}$ for (T, k) such that σ is a counterexample to $\mathcal{E}$ for (T, ω).*

To prove the above theorem, we need to show the existence of a sufficiently precise counter parameter k. We obtain such a k via the following result of [EFM99].

Upward Closure. For two $X \cup \{\delta\}$-states s, s', we say $s \leq s'$ if (1) for all $x \in X$, $s.x = s'.x$, and, (2) For each $q \in Q$, $s.\delta.q \leq s'.\delta.q$. The *upward closure* of set of $(X \cup \{\delta\})$-states G, written $G^{\leq}$, is the set $\{s' \mid \exists s \in G. s \leq s'\}$.

Lemma 1 ([EFM99]) *Let T be a finite-state thread, and $\mathcal{E}$ be a set of $T.X$-states. There exists a* finite *set of $(T.X \cup \{\delta\})$-states $G_{\mathcal{E}}$, such that:*

1. *For each $s \in G_{\mathcal{E}}$, for each $q \in Q$, we have $s.\delta.q \in \mathbb{N}$.*
2. *$s \in G_{\mathcal{E}}^{\leq}$ iff there exists $\{s'\} \lesssim \mathcal{E}$ and σ such that $s \overset{\sigma}{\rightsquigarrow}{}^{\omega} s'$.*

A corollary of the above lemma is that if $s' \in G_{\mathcal{E}}^{\leq}$ and $s \dot{\rightsquigarrow}^{\omega} s'$ then $s \in G_{\mathcal{E}}^{\leq}$. In other words, the set $G_{\mathcal{E}}^{\leq}$ is closed under the "predecessor" operation.

For a finite state thread T, and set of X-states $\mathcal{E}$, let $k_{\mathcal{E}}$ be defined using the set G above as:

$$k_{\mathcal{E}} = 1 + \max_{s \in G_{\mathcal{E}}} \max_{q \in Q} s.\delta.q$$

This is well defined as $G_{\mathcal{E}}$ is finite if T is finite-state. We shall show that if (T, ω) is safe w.r.t. $\mathcal{E}$, then $(T, k_{\mathcal{E}})$ is also safe w.r.t. $\mathcal{E}$. The main idea used in the proof is stated by the following lemma.

Lemma 2 *Let T be a finite-state thread, and $\mathcal{E}$ be a set of $T.X$-states. For every $s \dot{\rightsquigarrow}^{k_{\mathcal{E}}} s'$ if $s \notin G^{\leq}_{\mathcal{E}}$ then $s' \notin G^{\leq}_{\mathcal{E}}$.*

Proof. Suppose that $s \dot{\rightsquigarrow}^{k_{\mathcal{E}}} s'$. We shall prove the contrapositive, *i.e.,* if $s' \in G^{\leq}_{\mathcal{E}}$ then $s \in G^{\leq}_{\mathcal{E}}$. Assume that $s' \in G^{\leq}_{\mathcal{E}}$. Suppose that there exist r, r' such that (1) $r' \in G^{\leq}_{\mathcal{E}}$, (2) $r \dot{\rightsquigarrow}^{\omega} r'$, and, (3) $r \leq s$. Then (1),(2) coupled with the corollary of Lemma 1 implies that $r \in G^{\leq}_{\mathcal{E}}$, which together with (3) implies that $s \in G^{\leq}_{\mathcal{E}}$, and we are done.

Hence, it suffices to show the existence of an appropriate r, r'. For a counter map $\delta : Q \to \mathbb{N} \cup \{\omega\}$, and a location $q \in Q$, define the finitization $\lfloor \delta \rfloor_q : Q \to \mathbb{N}$ as:

$$\lfloor \delta \rfloor_q . q'' = \begin{cases} \delta . q'' & \text{if } \delta . q'' \in \mathbb{N} \\ k_{\mathcal{E}} + 1 & \text{if } q'' = q \\ k_{\mathcal{E}} & \text{o.w.} \end{cases}$$

Suppose that $s \overset{(q,l,q')}{\rightsquigarrow}{}^{k_{\mathcal{E}}} s'$. Then setting $r = s[\delta \mapsto \lfloor s.\delta \rfloor_q]$, and $r' = s'[\delta \mapsto \lfloor s'.\delta \rfloor_{q'}]$, we obtain r, r' with the above properties. Condition (1) holds as since $s' \in G^{\leq}_{\mathcal{E}}$ there exists some $t' \in G_{\mathcal{E}}$ s.t. $t' \leq s'$. For those q'' where $s'.\delta.q''$ is finite, $t'.\delta.q''$ is less than $s'.\delta.q''$, and for those q'' where $s'.\delta.q''$ is ω, the finitization keeps the value above that of the corresponding value of $t'.\delta$ as every element in the range of $t'.\delta$ is less than $k_{\mathcal{E}}$, by the definition of $k_{\mathcal{E}}$. We have (3) for the same reason. Notice that $r'.\delta.q = r.\delta.q - 1$, $r'.\delta.q' = r.\delta.q' + 1$, and, for all other q'', the $r'.\delta.q'' = r.\delta.q''$, and so (2) follows from the definitions of the transition relation for (T, ω). $\square$

Proof. (**of Theorem 11**). For the first case, we show that $(T, k_{\mathcal{E}})$ is safe w.r.t. $\mathcal{E}$.

We can generalize Lemma 2 by induction on the length of σ that for all σ, $s \overset{\sigma}{\rightsquigarrow}{}^{k_{\mathcal{E}}} s'$ implies that if $s \notin G^{\leq}_{\mathcal{E}}$ then $s' \notin G^{\leq}_{\mathcal{E}}$. Use Lemma 1 to see that if (T, ω) is safe w.r.t. $\mathcal{E}$, then $G^{\leq}_{\mathcal{E}}$ does not contain the initial state, *i.e.,* $(T, \omega).s_0 \notin G^{\leq}_{\mathcal{E}}$. Hence, $(T, k_{\mathcal{E}}).s_0$ (which is the same as $(T, \omega).s_0$) is not in $G^{\leq}_{\mathcal{E}}$. From the above generalization, $\mathsf{Reach}.(T, k_{\mathcal{E}}) \cap G^{\leq}_{\mathcal{E}} = \emptyset$. Lemma 1 implies that $G^{\leq}_{\mathcal{E}}$ contains $\mathcal{E}$, and so we can conclude that $(T, k_{\mathcal{E}})$ is safe w.r.t. $\mathcal{E}$.

For the second case, let σ be the shortest counterexample to $\mathcal{E}$ for (T, ω). Then setting $k = |\sigma| + 1$ suffices. It is easy to check that the sequence of states of (T, ω), corresponding

to σ, and leading to $\mathcal{E}$, there are always fewer than k threads at any location other than the initial location $T.s_0.pc$. Hence, the identical sequence, is a counterexample to $\mathcal{E}$ for (T, k). $\square$

We now give a counterexample-guided algorithm for the parameterized safety verification problem for finite-state multithreaded programs (Algorithm 10). The algorithm proceeds by iteratively refining the counter abstraction of the parameterized multithreaded program, until either the program is proved safe, or a genuine counterexample is found.

The procedure ModelCheck takes two arguments, a finite-state program P and a set of states $\mathcal{E}$ and checks whether P is safe w.r.t. $\mathcal{E}$, by iteratively constructing the set Reach.P until fixpoint, and checking if the result intersects $\mathcal{E}$. If so, it simply returns, and if not, it raises an exception which is the *shortest* P-trace that is an $\mathcal{E}$-counterexample. This procedure can be implemented using standard finite-state model checking techniques [CGP99]. The procedure to check if a counterexample is genuine simply compares the length of the trace with the counter parameter, more sophisticated refinement schemes can also be used. It is trivial to check that if the length of the trace is smaller than the counter parameter, then the trace corresponds to a trace of (T, ω), and hence, by Proposition 12, to a trace of ω^ω. The correctness and termination of the Algorithm 10 follow respectively from Proposition 12 and Theorem 11.

Algorithm 10 Counter-Guided Symmetric Verification

Require: Finite-state thread $T = (X \cup \{pc\}, \rightsquigarrow, \{s_0\})$, set of X-states $\mathcal{E}$.

```
1:  k := 0
2:  while true do
3:    try
4:      ModelCheck.(T, k).E
5:      return SAFE
6:    with (Exception(σ)) →
7:      if |σ| ≤ k then
8:        return UNSAFE(σ)
9:      else
10:       k := k + 1
11: done
```

Theorem 12 [Termination and Correctness] *When Algorithm 10 is given a finite-state thread T and a set of error states $\mathcal{E}$, it terminates. If it returns*

1. SAFE *then T^ω is safe w.r.t. $\mathcal{E}$.*

2. UNSAFE*(σ) then T^ω is not safe w.r.t. $\mathcal{E}$ and σ is a counterexample to $\mathcal{E}$ for (T, ω).*

It follows that for finite-state threads, if we use the trivial havocable abstraction structure, where each region is exactly a single state, then the algorithm 5.2.2 terminates.

5.8 Related Work

Data races are a major source of errors in concurrent programs. Race detection tools enable the construction of robust concurrent systems by finding, or confirming the absence of, races. The most popular approaches to the race detection problem, the "lock-set" and type-based methods, exploit the fact that many programs use *locks* to prevent data races.

In the lock-set approach, one attempts to either statically [DS91; Ste93; EA03], or dynamically[SBN+97; CLL+02], determine the set of locks that protect a variable, by tracking statically (by a data flow analysis) or dynamically (by instrumenting the program), the set of locks *always* held by the thread when accessing the variable, and checking that this set is not empty.

In the type-based approaches [FF01; BLR02], one expects the programmer to use type annotations to indicate which locks protect which variables, and then the type-system ensures that if the program type checks, then there are no races. Programmers, however, often use synchronization idioms that cause false positives for these tools (*i.e.,* the tool reports a possible race when there is none). Additionally, control flow information may be used for more precision [vPG03].

For programs with complex state-based synchronization idioms, both the above methods yield false positives, and one requires a more precise, path-sensitive analyis such as model checking. Software model checkers like SLAM [BR00] and BLAST [HJMS02] check sequential programs. Bandera [CDH+00], Java Pathfinder [HP00], Feaver [Hol00], and 3VMC [Yah01] check concurrent programs with a user supplied finite-state abstraction; the latter uses first-order predicates which, if appropriately chosen, can model dynamic creation; Verisoft [God97] and CMC [MPC+02] partially explore the concrete state space to find errors. Magic [COYC03] uses predicate abstraction and bisimulation minimization to check programs with finitely many threads communicating number of concurrent threads communicating by message-passing. Since communication is explicit in the model, abstraction and bisimulation

minimization can be done independently of the other threads, *i.e.,* reachability information is not required. Calvin [FQS02] requires that a suitable abstract context is provided, and then discharges the two verification tasks using ESC-based tools[FLL+02].

Our work is inspired by [Lub84] which is the earliest reference we found for using counters for symmetric programs, and the work of [EFM99] which we use for our completeness result; both these papers, as well as other research on *parameterized verification* [DRB02; BCR01] consider arbitrarily many threads, but assume a finite state abstraction for each thread is given.

Assume-Guarantee (or Rely-Guarantee) reasoning, has been studied by several authors [MC81; Jon83; AL91], as a way of decomposing large verification tasks. Recently, it has received renewed attention [AH99; McM00] and has led to many proofs of complex hardware designs well beyond the grasp of monolithic analyses [Eir98; HLQR99; JM01]. In recent work [FQS02], it has been applied to the verification of multithreaded programs. In all of the above, the appropriate environment assumptions must be supplied manually. It is similar to reasoning with pre- and post-conditions for procedures.

Bibliography

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[AIKY95] R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AM78] Tilak Agerwala and Jayadev Misra. Assertion graphs for verifying and synthesizing programs. Technical Report 83, University of Texas, Austin, 1978.

[And94] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BCC+02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation, Complexity, Analysis, Transformation: Essays Dedicated to Neil D. Jones*, Lecture Notes in Computer Science 2566, pages 85–108. Springer-Verlag, 2002.

[BCH+04a] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The blast query language for software verification. In *SAS 04: Static Analysis Symposium*, LNCS, pages XXX–YYY. Springer-Verlag, 2004.

[BCH+04b] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *ICSE 04: Software Engineering*, pages 326–335, 2004.

[BCR01] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multi-threaded software libraries. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031. Springer, 2001.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds. In *CAV 96: Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1996.

[BGS97] R. Bodik, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *PLDI 97: Programming Language Design and Implementation*, pages 146–158. ACM, 1997.

[BH99] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. *Theoretical Computer Science*, 221(1–2):211–250, 1999.

[BHJ+] D. Blei, C. Harrelson, R. Jhala, R. Majumdar, G. C. Necula, S. P. Rahul, W. Weimer, and D. Weitz. Vampyre: A proof generating theorem prover. http://www.eecs.berkeley.edu/~rupak/Vampyre.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA 02: Software Testing and Analysis*, pages 123–133. ACM, 2002.

[BLR02] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 02: Object-Oriented Programming, Systems, Languages and Applications*, pages 211–230, 2002.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.

[BPR01] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking c programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pages 268–283. Springer-Verlag, 2001.

[BPS00] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[BR] T. Ball and S.K. Rajamani. Personal communication. 2002.

[BR00] T. Ball and S. K. Rajamani. Boolean programs: a model and process for software analysis. Technical Report MSR Technical Report 2000-14, Microsoft Research, 2000.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001: SPIN Workshop*, LNCS 2057, pages 103–122. Springer-Verlag, 2001.

[BR02a] T. Ball and S.K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, 2002.

[BR02b] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV 94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer-Verlag, 1994.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.

[CCGS03] S. Chaki, E.M. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *CHARME 03: Correct Hardware Design and Verification*, LNCS 2860, pages 19–34. Springer, 2003.

[CDH+00] J. Corbett, M. Dwyer, John Hatcliff, Corina Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *ICSE 00: Software Engineering*, pages 439–448, 2000.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[CFR+91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadek. Efficiently computing static single assignment form and the program dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* Mit Press, 1999.

[Cla76] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions in Software Engineering*, 2(2):215–222, 1976.

[CLL+02] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI 2002: Programming Languages Design and Implementation*, pages 258–269. ACM, 2002.

[COYC03] S. Chaki, J. Ouaknine, K. Yorav, and E.M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *SoftMC 03: Software Model Checking*, 2003.

[Cra57] W. Craig. Linear reasoning. *J. Symbolic Logic*, 22:250–268, 1957.

[CW02] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM CCS 02: Conference on Computer and Communications Security*, pages 235–244. ACM, 2002.

[CWD02] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Usenix Security Symposium*, pages 171–190. Usenix, 2002.

[CYC+01] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system bugs. In *SOSP 01: ACM Symposium on Operating System Principles*, pages 78–81. ACM Press, 2001.

[DC94] M. Dwyer and L. Clarke. Data flow analysis for verifying properties of concurrent programs. In *FSE 94: Foundations of Software Engineering*, pages 62–75. ACM, 1994.

[DD01] S. Das and D.L. Dill. Successive approximation of abstract transition relations. In *LICS 01: Logic in Computer Science*, pages 51–60. IEEE Press, 2001.

[DD02] S. Das and D.L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD 02: Formal Methods in Computer-Aided Design*, LNCS 2517, pages 19–32. Springer, 2002.

[DDP99] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV 99: Computer-Aided Verification*, LNCS 1633, pages 160–171. Springer-Verlag, 1999.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[DNS] D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem prover. http://research.compaq.com/SRC/esc/Simplify.html.

[DRB02] G. Delzanno, J.-F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded java programs. In *TACAS 02: Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–187. Springer, 2002.

[DS91] Annette Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[EA03] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *SOSP 03: ACM Symposium on Operating System Principles*, pages 237–252. ACM Press, 2003.

[ECCH00] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association, 2000.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS 99: Logic in Computer Science*, pages 352–359. IEEE Press, 1999.

[Eir98] Asgeir P. Eiriksson. The formal design of 1m-gate asics. In *FMCAD 98: Formal Methods in Computer-Aided Design*, LNCS 1522, pages 49–63. Springer-Verlag, 1998.

[Ern00] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, Seattle, 2000.

[Eva96] David Evans. Static detection of dynamic memory errors. In *PLDI*, pages 44–53, 1996.

[FD04] M.A. Fahndrich and R. DeLine. Typestates for objects. In *ECOOP 04: Object-Oriented Programming*, LNCS 3086, pages 465–490. Springer, 2004.

[FF01] C. Flanagan and S.N. Freund. Detecting race conditions in large programs. In *PASTE 01: Program Analysis for Software Tools and Engineering*, pages 90–96. ACM, 2001.

[FFA99] J. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *PLDI 99: Programming Languages Design and Implementation*, pages 192–203. ACM, 1999.

[FIS00] A. Finkel, S. P. Iyer, and G. Sutre. Well-abstracted transition systems. In *CONCUR 00: Concurrency Theory*, LNCS 1877, pages 566–580. Springer-Verlag, 2000.

[FLL+02] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[Flo67] R.W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[FORS01] J.-C. Filliâtre, S. Owre, H. Ruess, and N. Shankar. ICS: Integrated canonizer and solver. In *CAV 01: Computer-aided verification*, LNCS, pages ???–??? Springer-Verlag, 2001.

[FQ02] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL 02: Principles of Programming Languages*, pages 191–202. ACM, 2002.

[FQS02] C. Flanagan, S. Qadeer, and S.A. Seshia. A modular checker for multithreaded programs. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 180–194. Springer, 2002.

[FS01] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL 01: Principles of Programming Languages*, pages 193–205. ACM, 2001.

[FTA01] J.S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. Technical Report CSD-01-1162, University of California, Berkeley, 2001.

[FTA02] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.

[GBR02] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA 98: Software Testing and Analysis*, pages 53–62. ACM, 2002.

[GLvB+03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI 2003: Programming Languages Design and Implementation*, pages 1–11. ACM, 2003.

[GMS98] N. Gupta, A. Mathur, and M.L. Soffa. Generating test data for branch coverage. In *ASE 00: Automated Software Engineering*, pages 219–228. IEEE, 1998.

[God97] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186. ACM, 1997.

[GP02] E. Gunter and D. Peled. Temporal debugging for concurrent systems. In *TACAS 02: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 431–444. Springer, 2002.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[GS97a] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[GS97b] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[HCL+03] H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE 2003: Software Engineering*, pages 232–243. ACM, 2003.

[HHK95] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36rd Annual Symposium on Foundations of Computer Science*, pages 453–462. IEEE Computer Society Press, 1995.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[HJM+02] T.A. Henzinger, R. Jhala, R. Majumdar, G.C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 526–538. Springer, 2002.

[HJM04] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI 2004: Programming Languages Design and Implementation*, pages 1–12. ACM, 2004.

[HJMM04] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.

[HJMQ03] T.A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV 03: Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[HJMS02] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.

[HJMS03] T.A. Henzinger, R. Jhala, R. Majumdar, and M.A.A. Sanvido. Extreme model checking. In *International Symposium on Verification*, LNCS. Springer, 2003.

[HLQR99] T.A. Henzinger, X. Liu, S. Qadeer, and S.K. Rajamani. Formal specification and verification of a dataflow processor array. In *Proceedings of the International Conference on Computer-aided Design*, pages 494–499. IEEE Computer Society Press, 1999.

[HM00] T. A. Henzinger and R. Majumdar. A classification of symbolic transition systems. In *STACS 00: Theoretical Aspects of Computer Science*, LNCS 1770, pages 13–34. Springer-Verlag, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hol00] G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In *SPIN 00: Spin Model Checking and Software Verification*, LNCS 1885, pages 131–147. Springer, 2000.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java Pathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.

[HSW+00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS 2000: Architectural Support for Programming Languages and Operating Systems*, pages 93–104. ACM, 2000.

[JBW+94] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and infeasible path analysis. In *ISSTA 94: Software Testing and Analysis*, pages 95–107. ACM, 1994.

[JM01] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In *CAV 01: Computer Aided Verification*, LNCS 2102, pages 396–410. Springer, 2001.

[Jon83] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[JV00] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA 00: Software Testing and Analysis*, pages 14–25. ACM, 2000.

[JW04] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, pages 119–134. USENIX, 2004.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206, 1973.

[Kin76] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KMM00] M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[KPV03] S. Khurshid, C.S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS 03: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, pages 553–568. Springer, 2003.

[Kra97] J. Krajicek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symbolic Logic*, 62:457–486, 1997.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[LAS00] T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS 02: Static Analysis Symposium*, Lecture Notes in Computer Science 2280, pages 280–301. Springer-Verlag, 2000.

[Lub84] B.D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs i. *Acta Informatica*, 21:125–169, 1984.

[Man69] Z. Manna. The correctness of programs. *Journal of Computer and Systems Sciences*, 3(2):119–127, 1969.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[McM93] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem*. Kluwer Academic Publishers, 1993.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 24–35. Springer, 1997.

[McM00] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37((1–3)):279–309, 2000.

[McM03] K.L. McMillan. Interpolation and SAT-based model checking. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 1–13. Springer, 2003.

[McM04] K.L. McMillan. An interpolating theorem prover. In *TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2988, pages 16–30. Springer, 2004.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC 01: Design Automation Conference*, pages 530–535, 2001.

[Mor82] J. M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, Lecture Notes of an International Summer School, pages 25–34. D. Reidel Publishing Company, 1982.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proc. Symposia in Applied Mathematics*. American Mathematical Society, 1967.

[MPC+02] M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation*. ACM, 2002.

[Mye79] G.J. Myers. *The Art of Software Testing*. Wiley, 1979.

[Nam01] K. Namjoshi. Certifying model checkers. In *CAV 01: Computer Aided Verification*, LNCS 2102, pages 2–13. Springer, 2001.

[Nec97a] G.C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[Nec97b] G.C. Necula. Proof carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.

[Nel81] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[NL98] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Thirteenth Annual Symposium on Logic in Computer Science*, pages 93–104, Indianapolis, June 1998. IEEE Computer Society Press.

[NMRW02] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, Lecture Notes in Computer Science 2304, pages 213–228. Springer, 2002.

[NR01] G.C. Necula and S.P. Rahul. Oracle-based checking of untrusted software. In *POPL 01: Principles of Programming Languages*, pages 142–154. ACM, 2001.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, 1997.

[Pel01] D. Peled. *Software reliability methods*. Springer, 2001.

[Pel03] D. Peled. Model checking and testing combined. In *ICALP 2003: Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 47–63. Springer, 2003.

[Pfe97] F. Pfenning. Computation and deduction. Lecture notes, 1997.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[Pud97] P. Pudlak. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62:981–998, 1997.

[PY03] M. Pezze‘ and M. Young. Software test and analysis: Process, principles, and techniques. Manuscript, 2003.

[PZ01] D. Peled and L.D. Zuck. From model checking to a temporal proof. In *SPIN 2001: SPIN Workshop*, Lecture Notes in Computer Science 2057, pages 1–14. Springer-Verlag, 2001.

[QS81] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Fifth International Symposium on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1981.

[RHC76] C. Ramamoorthy, S.B. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions in Software Engineering*, 2(2):293–300, 1976.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[RSY04] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In Bernhard Steffen and Georgio Levi, editors, *VMCAI*

2004: Verification Model Checking and Abstract Interpretation, LNCS 2937, pages 252–266. Springer-Verlag, 2004.

[Sai00] H. Saidi. Model checking guided abstraction and analysis. In *SAS 00: Static-Analysis Symposium*, pages 377–396. LNCS 1824, Springer-Verlag, 2000.

[SBD02] A. Stump, C.W. Barrett, and D.L. Dill. CVC: A cooperating validity checker. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 500–504. Springer, 2002.

[SBN+97] S. Savage, M. Burrows, C.G. Nelson, P. Sobalvarro, and T.A. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[Som98] F. Somenzi. Colorado university decision diagram package. http://vlsi.colorado.edu/pub/, 1998.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data dalow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV 99: Computer-aided Verification*, LNCS 1633, pages 443–454. Springer-Verlag, 1999.

[Ste93] N. Sterling. Warlock: a static data race analysis tool. In *USENIX Winter 1993 Technical Conference*, pages 97–106, 1993.

[SY86] R.E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[Tur36] Alan M. Turing. On computable numbers, with an application to the eintscheidungsproblem. In *Proceedings of the London Mathematical Soceity*, pages 230–265, 1936.

[USW01] Jeffrey Foster Umesh Shankar, Kunal Talwar and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, 2001.

[vPG03] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI 2003: Programming Languages Design and Implementation*, pages 115–128. ACM, 2003.

[Wad90] Philip Wadler. Linear types can change the world! In *IFIP TC2 Working Conference on Programming Concepts and Methods*, 1990.

[Win93] G. Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, 1993.

[Yah01] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL 01: Principles of Programming Languages*, pages 27–40. ACM Press, 2001.