Avoiding State-Space Explosion in Multithreaded Programs with *Input-Covering Schedules* and *Symbolic Execution*

Tom Bergan

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

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Luis Ceze, Chair

Daniel Grossman, Chair

Michael Ernst

Program Authorized to Offer Degree:
Department of Computer Science & Engineering, University of Washington

University of Washington

Abstract

Avoiding State-Space Explosion in Multithreaded Programs with *Input-Covering Schedules* and *Symbolic Execution*

Tom Bergan

Co-Chairs of the Supervisory Committee:
Associate Professor Luis Ceze
Department of Computer Science & Engineering

Associate Professor Daniel Grossman
Department of Computer Science & Engineering

This dissertation makes two high-level contributions:

First, we propose an algorithm to perform *symbolic execution* of multithreaded programs from *arbitrary* program contexts. We argue that this can enable more efficient symbolic exploration of deep code paths in multithreaded programs by allowing the symbolic engine to jump directly to program contexts of interest. We are the first to attack this problem.

Second, we propose constraining multithreaded executions to small sets of *input-covering schedules*, which are defined as follows: given a program P, we say that a set of schedules Σ *covers* all inputs of program P if, when given any input, P's execution can be constrained to some schedule in Σ and still produce a semantically valid result. Our approach is to first compute a small Σ for a given program P, and then, at runtime, constrain P's execution to always follow some schedule in Σ, and never deviate. This approach has the following advantage: because all possible runtime schedules are known *a priori*, we can seek to validate the program by thoroughly verifying each schedule in Σ, in isolation, without needing to reason about the huge space of thread interleavings that arises due to conventional nondeterministic execution.

To tie both contributions together, we show how our symbolic execution techniques can be used to speed the search for input-covering schedules.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Although the work described herein was performed primarily by myself, I did not work in a vacuum. Much of my work grew directly from collaborations with others, and in recognition of those collaborations, I will use the pronoun "we" throughout the main body of this document.

I am indebted to my advisors: Luis Ceze brought enthusiasm and a non-stop deluge of ideas. He balanced my inner cynicism with optimism, and helped me find the good ideas in my work, especially when I thought there weren't any good ideas to find. Steve Gribble brought a similar sense of optimism and encouragement. Dan Grossman brought a knack for devising concise explanations of complex topics: many of the organizational strategies, turns of phrase, and formal semantics that I use in this document are based directly on his suggestions. Mike Ernst's pointed questions during practice talks helped me learn to chose words carefully, and also helped me view my work from a different perspective. Eddie Kohler got me started in computer science research: he provided an encouraging environment that was intellectually stimulating and fun, and my experience working in his research group convinced me that getting a Ph.D. would be enjoyable.

I am particularly grateful to have had the opportunity to dabble in multiple subfields of computer science: computer architecture, operating systems, distributed systems, compilers, and program analysis. This diversity of experience has been the most intellectually fulfilling part of graduate school and it is a direct result of the open culture of UW CSE combined with the flexibility and open-mindedness of the advisors that I've been lucky to work with.

My fellow graduate students at UW have made my time here more rewarding. I'm grateful for support from UW's SAMPA and PLSE groups. Much of my work has grown directly from conversations and debates with Joe Devietti, and his feedback and ideas have greatly improved my projects. Owen Anderson showed me the ropes in LLVM. It was fun to code with Nicholas Hunt, who helped with much of the coding in dOS [14] (a major project that is not the subject of this dissertation). My officemate Brandon Lucia was a constant source of constructive feedback, encouragement, and fun technical discussions. Jacob Nelson provided a wealth of random technical and historical knowledge, and worked almost single-handedly to keep our lab's machines running. Adrian Sampson, Ben Wood, and Colin Gordon sat through many practice talks and helped edit many paper drafts.

Just as research ideas do not spring from a void, neither do research artifacts. I've made use of many publicly available tools and libraries that were released by very generous people. I especially want to acknowledge the authors of LLVM [68], KLEE [27], Cloud9 [24], STP [46], and DSA [67]—these tools are used directly in my dissertation work—and I also want to acknowledge the authors of the benchmark suites SPLASH2 [95] and PARSEC [16], as benchmark creation is an extremely important but largely thankless endeavor.

Lastly, to all of you who've made my life enjoyable outside of work and school: thank you! This includes, but is not limited to, my parents and sisters, my extended family, and my friends in and around Seattle, especially Molly Douglas. Through good jokes, dumb jokes, outdoorsy adventures, and camaraderie, you keep me sane.

Chapter 1

OVERVIEW

Multithreaded programs are notoriously difficult to test and verify. In addition to the already daunting task of reasoning about program behavior over all possible inputs, testing and verification tools must reason about a large number of possible thread interleavings for each input, as the number of possible interleavings grows exponentially with the length of a program's execution. Tools can systematically explore the thread interleaving space in part, but in practice, the interleaving space is so massive that it cannot be explored exhaustively [25, 77].

How can we reason about the behavior of multithreaded programs when the thread interleaving space is so enormous? Many researchers have proposed partial answers to this question. This dissertation proposes two entirely new approaches. Our new approaches are not perfect (no approach is), but they provide unique benefits. To summarize:

- Our first approach is to perform *symbolic execution* of multithreaded programs starting from *arbitrary* program contexts. We argue that this can enable more efficient symbolic exploration of deep code paths in multithreaded programs by allowing the symbolic engine to jump directly to program contexts of interest.

- Our second approach is to constrain execution to small sets of *input-covering schedules*. By constraining execution to small sets of schedules, the problems of *testing* and *verification* become simpler, as testing and verification tools no longer need to reason about an enormous interleaving space.

Our two approaches are complementary to each other and to prior approaches. Notably, we show how our symbolic execution techniques enable our algorithm for enumerating input-covering schedules. Our work in this dissertation focuses on multithreaded programs that are written in the C language using the POSIX threads library (pthreads) [54]—we chose to focus on this combination of language and library due to its ubiquity—although many of our techniques can be applied more generally.

The remainder of this chapter provides context for the rest of this dissertation. First, we provide a brief background on the multithreaded programming model (Section 1.1 and Section 1.2). Then, we summarize our contributions (Section 1.3 and Section 1.4) and give an outline for the rest of this dissertation (Section 1.5).

1.1 Shared-Memory Multithreading

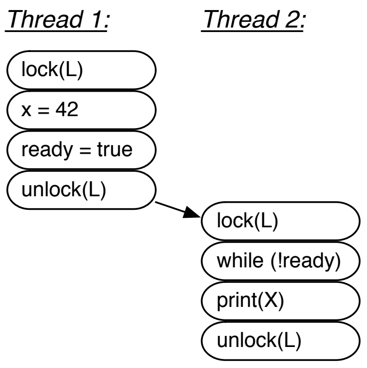
This dissertation focuses on *shared-memory multithreaded programs* written in the C language. In this programming model, each program is composed of multiple threads of control. Each thread executes its own instruction sequence, and threads communicate by reading and writing values from and to a shared memory space.

The multithreading programming model has two primary uses. First, by mapping different threads to different hardware CPUs, many threads can execute simultaneously, leading to faster overall execution. This is known as *parallelism*. Second, even when only one hardware CPU is available, a programmer might assign logically simultaneous tasks to separate threads. For example, a database server might spawn a separate thread for each request, where those threads use shared memory to coordinate updates to the database. This is known as *concurrency*. Shared-memory multithreading is not the only programming model for parallelism and concurrency (cf. [2, 3, 7, 91]), but it is a popular model that is supported by many widely-used programming languages, including C, C++, Java, and C#.

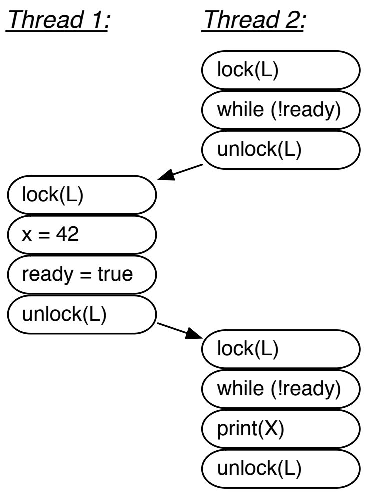
Abstractly, we model shared memory as a map from locations to values, and we model each thread's execution as a sequence of reads and writes that operate on shared memory. Globally, operations from multiple threads are interleaved in some way. The challenge is that this interleaving is *nondeterministic*. For example, suppose location $X = 0$ in shared memory, and suppose that thread T_1 writes the value 42 to location X while thread T_2 reads location X. Which value is read by T_2 depends on the interleaving of these threads, which in turn depends on low-level timing variations in the hardware and the operating system (OS), such as the relative clock rate of each CPU, scheduling decisions made by the cache

```
global int X = 0
global bool ready = false
global lock L

Thread 1             Thread 2
lock(L)              lock(L)
x = 42               while (!ready) {
ready = true            unlock(L)
unlock(L)               lock(L)
                     }
                     print(X)
                     unlock(L)
```



(a)



(b)

Figure 1.1: Examples of *happens-before* graphs, demonstrating that programs can execute with nondeterministic interleavings even when they produce deterministic results. Happens-before graphs (a) and (b) illustrate two possible executions of the program at the top of this figure. In each graph, there are implicit *program order* edges connecting the adjacent nodes in each thread (*e.g.*, there is an implicit edge from `lock(L)` to `x=42` in thread T_1). No matter which schedule is chosen, the program always prints "42".

hardware, and scheduling decisions made by the OS. Hence, if the program is executed repeatedly, T_2 can read the value 0 in some executions and 42 in other executions.

The programmer can restrict interleavings by using language-provided synchronization primitives such as locks, barriers, and condition variables. Figure 1.1 shows how locks can be used to ensure that T_2 always reads 42. In this example, because T_2 reads the value written by T_1, we say that T_1's write "happens before" T_2's read. This derives from the *happens-before* relation [65], which is the irreflexive transitive closure of program order (the single-threaded order in which operations statically appear in the program) and synchronization order (the order imposed by synchronization during an execution). We often draw the happens-before relation as a directed graph in which operation A happens-before B iff there exists a directed path from A to B in the happens-before graph. In this dissertation, we sometimes refer to happens-before graphs as *schedules*.

Even though the program in Figure 1.1 produces a deterministic final state, operations are still interleaved nondeterministically during execution. This is demonstrated by Figure 1.1(a) and Figure 1.1(b), which illustrate two possible happens-before graphs that can result from different executions of this program.

The set of possible interleavings is further restricted by the language's memory model. The simplest model is *sequential consistency* [66], in which operations from all threads are serialized into some total order. Since all work in this dissertation targets the C language, we adhere to the memory model specified for C [17, 55]. Briefly, C ensures sequentially consistent execution for data race free programs (defined below), but provides undefined semantics for programs with data races.

1.2 The Dark Side of Nondeterminism

We have seen that multithreaded programs execute nondeterministically. Unfortunately, this nondeterminism leads to a variety of insidious bugs. The most notorious is a *data race*, which occurs when two threads access the same location, where the accesses are concurrent (neither happens-before the other) and at least one access is a write. We consider *every* data race an error in C [19, 20].

Figure 1.2 illustrates a data race: the accesses of X in thread T_1 are not ordered with

Data Race
```
Thread 1       Thread 2
tmp = X
               X = 42
X = tmp+1
```

Deadlock
```
Thread 1       Thread 2
lock(A)
               lock(B)
lock(B)        lock(A)
```

Atomicity Violation
```
Thread 1       Thread 2
lock(L)
tmp = X
unlock(L)      lock(L)
               X = 42
lock(L)        unlock(L)
X = tmp+1
unlock(L)
```

Assertion Failure
```
Thread 1       Thread 2
               lock(L)
               assert(X%2!=0)
lock(L)        unlock(L)
if (X%2==0)
  X++
unlock(L)
```

Figure 1.2: Examples of concurrency bugs that manifest nondeterministically in multithreaded programs. The relative spacing of statements describes interleavings under which each bugs manifests. The *assertion failure* example is buggy when `X` is initially even.

the write of X in thread T_2 due to the lack of synchronization. Hence, T_1 will not correctly increment X if T_2's write happens to interleave between T_1's two statements. This race can be removed by guarding each statement with a lock—although such use of locks technically avoids the data race, it does not fix the underlying defect as the buggy interleaving is still possible, this time as an *atomicity violation.* Figure 1.2 illustrates this atomicity violation, and also illustrates two other kinds of bugs that are common in multithreaded programs: a *deadlock* and an ordering-dependent *assertion failure.*

The Challenge. Ideally, we would use thorough testing and verification to either find all data races, atomicity violations, deadlocks, and assertion failures, or prove their absence. Unfortunately, each of these bugs is triggered on some *subset* of executions, only. For example, in Figure 1.2, if T_2 happens to execute entirely before or entirely after T_1, then the atomicity violation is quietly avoided. Thus, the atomicity violation in Figure 1.2 manifests only on a specific, unlucky choice of schedule. Further, each schedule *might* manifest a bug—it is difficult to tell whether or not a schedule manifests a bug without analyzing the schedule directly. Given a program with n threads, where each thread uses an average of k synchronization operations per execution, there are $O(n! \cdot k!)$ possible schedules, and any of them can be buggy. With so many possible schedules, how can multithreaded programs be thoroughly tested and verified?

1.3 Reasoning about Multithreaded Programs with Symbolic Execution

The most naïve way to analyze a multithreaded program is to perform a brute-force exploration of all feasible thread schedules and execution paths. *Symbolic execution* performs this brute-force exploration systematically. The idea is to execute programs with symbolic rather than concrete inputs and use an SMT (SAT Modulo Theory) solver to prune infeasible paths. On branches with more than one feasible resolution, the symbolic state is forked and all feasible resolutions are explored. The key advantage of this approach is precision—unlike techniques such as abstract interpretation [33], symbolic execution is generally free of false positives because its semantics are fully precise up to the limits of the underlying SMT solver. Recent advances in SMT solving have made symbolic execution faster and more practical [46, 76], paving the way for recent systems that have used symbolic execution with great success [24, 27, 30, 49, 50, 60, 92].

The chief difficulty is path explosion: the number of feasible execution paths is generally exponential in the length of an execution, and this number grows even larger when symbolic execution is applied to multithreaded programs [24, 60]. As described above, multithreaded programs suffer from an explosion of possible thread interleavings in addition to the explosion of single-threaded paths. Prior work has dealt with path explosion with a variety of approaches that we detail in Section 4.1.

Our approach is to limit path explosion by symbolically executing relatively small fragments of a program in isolation—this reduces path length, which in turn reduces the potential for path explosion. Prior work has largely assumed that symbolic execution will begin at one of a few natural starting points, such as program entry (for whole-program testing) or a function call (for single-threaded unit testing). We do not make such an assumption—we allow program fragments to begin anywhere—so our main challenge is to perform symbolic execution of multithreaded programs from *arbitrary* program contexts.

Problem Statement. Specifically, we address the following problem: given an *initial program context*, which we define to be a set of threads and their program counters, how do we efficiently perform symbolic execution starting from that context while soundly accounting for all possible concrete memory states at that initial program context?

Solution Overview. We solve this problem in two parts. First, we use a *context-specific dataflow analysis* to construct an over-approximation of the initial state for the given program context. Second, we integrate that analysis with a novel *symbolic execution semantics* that can execute forward from an abstract initial state, even when precise information about pointers and synchronization is not available. Our solution makes a tradeoff: approximating the initial state results in some loss of precision, but enables high-coverage analysis in deep program paths that were previously unreachable.

Contributions. We are the first to attack this problem. Additionally, our contributions include:

- An algorithm for performing symbolic execution from arbitrary program contexts. The novel features of this algorithm include: (a) a way to integrate conservative dataflow analyses with symbolic execution to increase symbolic precision at arbitrary program contexts, and (b) a novel semantics for reasoning symbolically about concurrency.

- An implementation of that algorithm and an empirical evaluation of our implementation on a range of realistic programs. Our implementation supports C programs that use pthreads. We find that our integration of dataflow analysis and symbolic execution is *vital* for preserving a reasonable level of precision.

1.4 Restricting the Schedule Space with Input-Covering Schedules

Our approach to symbolic execution limits path explosion by focusing on small fragments of execution. However, it does not attack the most fundamental problem: multithreaded programs can still execute under an enormous number of thread schedules.

Do we actually need all of those schedules? We believe the answer is *no*. Hence, we propose constraining multithreaded execution to small sets of *input-covering schedules*. Given a program P, we say that a set of schedules Σ *covers* the program's inputs if, for all inputs, there exists some schedule $S \in \Sigma$ such that P's execution can be constrained to S and still produce a semantically valid result.

Our Approach. We propose the following deployment strategy. First, given a program P, we enumerate a set of input-covering schedules Σ for program P using an algorithm based on symbolic execution. Each schedule in Σ is paired with an *input constraint* that describes the set of inputs under which the schedule can be followed. Each schedule is specified as a partial order of dynamic instances of synchronization statements, *i.e.*, each schedule is a happens-before graph.

Second, we deploy P along with a custom runtime system that constrains execution of P to follow schedules in Σ *only.* Our custom runtime system captures the program's inputs, find a pair (I,S) $\in \Sigma$ such that the program's inputs satisfy input constraint I, and then constrain execution to S, ensuring that execution never deviates from S. Finally, and most importantly, given that all executions of P will be constrained to schedules in Σ, testing and verification becomes simpler—the input-covering set Σ contains the *complete* set of schedules that might be followed at runtime, and as a result, testing and verification tools can focus on schedules in Σ only, avoiding the need to reason about the massive space of possible interleavings.

Crucially, our approach assumes that it is possible to enumerate small sets of input-covering schedules to begin with. It is not obvious that small sets of input-covering schedules should exist for realistic multithreaded programs. The key word is *small*—an input-covering set Σ is of no help when it is so intractably large that it cannot be enumerated in a reasonable time. An important contribution of this work is defining Σ in a way that makes the problem more tractable. Notably, programs that run for unbounded periods of time can require unboundedly many schedules, making the set Σ intractably large. We avoid this problem by partitioning execution into *bounded epochs*—we find input-covering schedules for each epoch in isolation, and then piece those schedules together at runtime. Bounded epochs themselves introduce technical complexities, and it is exactly these complexities where our new symbolic execution techniques will become useful.

Contributions. We are the first to propose the concept of input-covering schedules, and the first to propose a framework for exploiting them. Additionally, our contributions include:

- An algorithm for enumerating input-covering schedules.

- An implementation of that algorithm, along with an empirical evaluation of the implementation on a range of realistic programs. Our implementation supports C programs that use pthreads. Our evaluation characterizes cases in which the algorithm works well, as well as cases in which it does not work well.

- An implementation of a runtime system for constraining execution to a set of input-covering schedules. In this work, we aim for a proof-of-concept rather than fully-optimized implementation.

- A simple deadlock checker that exploits input-covering schedules. In this work, we aim to demonstrate how input-covering schedules can simplify the process of building such a checker—we do not aim to produce a feature-complete checker.

1.5 Outline

Chapter 2 presents our approach to symbolic execution of multithreaded programs from arbitrary program contexts. We describe our dataflow semantics, our symbolic execution semantics, our implementation, and our empirical evaluation. Proofs of soundness and completeness are given in appendices at the end of this dissertation.

Chapter 3 presents our algorithm for enumerating a set of input-covering schedules. We also describe our implementation, a proof-of-concept runtime system and deadlock checker, and our empirical evaluation.

Chapters 2 and 3 are updated and expanded versions of papers originally published in other venues [11, 13]. Chapter 4 surveys related work. Chapter 5 gives concluding remarks and discusses the potential for future work.

Chapter 2

SYMBOLIC EXECUTION FROM ARBITRARY PROGRAM CONTEXTS

This chapter describes an analysis that can perform symbolic execution of a multithreaded program starting from an arbitrary program context. We start by restating the problem and summarizing our solution (Section 2.1). We further motivate our work in this chapter by describing ways in which our analysis is useful (Section 2.2).

Next, we explain the mechanics of our approach. Our goal is to analyze multithreaded C programs, so we define our analysis over a simple core language called *SimpThreads* that maintains the features of C that make our analysis challenging. Notably, our core language includes explicit threads and synchronization, shared memory, and pointers that can refer to the interior of an object via pointer arithmetic. To simplify the exposition, we organize our explanation by language feature. We start with a simple, single-threaded subset of *SimpThreads* that has no pointers (Section 2.3). We then add pointers (Section 2.4) and threads (Section 2.5). At each step, we explain how we overcome the challenges introduced by each additional language feature. After giving this exposition, we give a summarized and cohesive description of our approach (Section 2.6).

Next, we state soundness and completeness theorems (Section 2.7). We have implemented our analysis on top of Cloud9 [24], which is a symbolic execution engine for C programs that was developed by other researchers. Cloud9 is in turn based on the widely-used KLEE [27]. We discuss our implementation (Section 2.8) and end by discussing an empirical evaluation of that implementation (Section 2.9).

2.1 Problem Statement and Overview

Our goal is scalable symbolic execution of multithreaded programs written in the C language and its derivatives. Our approach is to limit path explosion by symbolically executing relatively small fragments of a program in isolation—this reduces path length, which in turn reduces the potential for path explosion. Rather than exploring ways that program fragments might be selected, our work focuses on a more basic question: how do we symbolically execute a fragment of a multithreaded program in isolation, *soundly* and *efficiently*? Prior work has largely assumed that symbolic execution will begin at one of a few natural starting points, such as program entry (for whole-program testing) or a function call (for single-threaded unit testing). We do not make such an assumption—we assume that program fragments can begin anywhere—so our main challenge is to perform symbolic execution of multithreaded programs from *arbitrary* program contexts.

Specifically, we address the following problem: given an *initial program context*, which we define to be a set of threads and their program counters, how do we efficiently perform symbolic execution starting from that context while soundly accounting for all possible concrete initial states? We solve this problem in two parts. First, we use a *context-specific dataflow analysis* to construct an over-approximation of the initial state for the given program context. Second, we integrate that dataflow analysis with a novel *symbolic execution semantics* that can execute forward from an abstract initial state, even when precise information about pointers and synchronization is not available.

Constructing an Initial State. The most precise strategy is to symbolically execute all paths from program entry to the initial context, and then use path merging [52] to construct an initial state. This is not scalable—it suffers from exactly the sort of path explosion problems we are trying to avoid. Instead, we must *approximate* the initial state. The least precise approximation is to leave the initial state completely unconstrained, for example by assigning a fresh symbolic variable to every memory location. This is too conservative—it covers many memory states that never occur during any actual execution—and as a result, symbolic execution would investigate many infeasible paths.

Our solution represents a middle ground between the above two extremes: we use a *context-specific* dataflow analysis to construct a sound *over-approximation* of the initial state. We use an over-approximation to ensure that all feasible concrete initial states are included. Our dataflow analysis infers constraints on the initial memory state as well as constraints on synchronization, such as *locksets*, that together help symbolic execution avoid

```
 1  global int X,Y
 2  global struct Node { Lock lock, int data } nodes[]
 3
 4  Thread 1                   Thread 2
 5    void RunA() {              void RunB() {
 6      i = ...                    k = ...
 7      Foo(&nodes[i])             Bar(&nodes[k])
 8    }                          }
 9    void Foo(Node *a) {        void Bar(Node *b) {
10      for (x in 1..X) {          lock(b->lock)
11  ⇒     lock(a->lock)      ⇒   for (y in 1..Y)
12        ...                        ...
```

Figure 2.1: A simple multithreaded program that illustrates the challenges of beginning symbolic execution at an arbitrary program context. Especially notable are challenges that arise from explicit synchronization and from C-like pointers.

infeasible paths.

To illustrate, suppose we are asked to begin symbolic execution from the program context marked by arrows in Figure 2.1. This context includes two threads, each of which is about to execute line 11. Can lines 11 and 12 of `Foo` execute concurrently with lines 11 and 12 of `Bar`? To answer this question, we must first answer a different question: does thread t_2 hold any locks at the beginning of the program context (*i.e.*, at line 11)? Here we examine the locksets embedded in our initial state and learn that t_2 holds lock `b->lock`. Next, we ask a second question: does `a==b`? Suppose our dataflow analysis determines that `i==k` at line 6, and that `Foo` and `Bar` are called from `RunA` and `RunB` only. In this case, we know that `a==b`, which means that line 11 of `Foo` cannot execute concurrently with line 11 of `Bar`.

Symbolic Execution Semantics. The input to symbolic execution is an abstract initial state constructed by our dataflow analysis. The output is a set of pairs (`path`, `C`), where `path` is a path of execution and `C` describes a *path constraint* such that when `C` is satisfied on the initial state, the `path` can be followed. To support multithreaded programs, we make each `path` a serialized (sequentially consistent) trace of a multithreaded execution.

The key novelty of our symbolic semantics is the way it integrates with our dataflow analysis. For example, we exploit locksets, as described above, along with other invariants to improve the precision of various *symbolic synchronization* primitives. We reason about the initial values of local variables by exploiting *reaching definitions* that our dataflow analysis computes. We additionally exploit a static points-to analysis to help reason about aliasing relationships between pairs of *symbolic pointers*. Notably, our semantics can reason about symbolic pointers that may refer to the interior bytes of an object.

However, our dataflow analysis is necessarily conservative. It may leave portions of the memory state unconstrained, leaving us unable to precisely answer simple questions such as "which object does pointer X refer to?" in all cases. For example, suppose our dataflow analysis cannot determine if `i==k` at line 6. In this case, we must investigate two paths during symbolic execution: one in which `a==b`, and another in which `a!=b`. For this reason, the set of paths explored by symbolic execution may be a *superset* of the set of paths that are actually feasible.

Soundness and Completeness. Our symbolic semantics are sound and complete up to the limits of the underlying SMT solver. By *sound*, we mean that if our symbolic execution outputs a pair (`path`, `C`), then, from every concrete initial state that satisfies constraint `C`, concrete execution *must* follow `path` as long as context switches are made just as in `path`. By *complete*, we mean that symbolic execution outputs a set of pairs (`path`, `C`) sufficient to cover *all* possible concrete initial states that may arise during any valid execution of the program.

SMT solvers are incomplete in practice—they may timeout on a difficult query, or they may not support all kinds of expressions available in the target language. Hence, our symbolic semantics are incomplete in practice. Further, even if a complete SMT solver were available, our implementation is still incomplete in practice as there may be *too many* feasible paths—the number of feasible paths typically grows exponentially with the length of execution, so any large, realistic program may have too many feasible paths to be practicably enumerable in any reasonable time.

2.2 Applications

The analysis presented in this chapter has a variety of promising applications:

Focused Testing of Program Fragments. We can test an important parallel loop in the context of a larger program. Classic symbolic execution techniques require executing deep code paths from program entry to reach the loop in the first place, where these deep paths may include complex initialization code or prior parallel phases. Our techniques enable testing the loop directly, using a fast and scalable dataflow analysis to summarize the initial deep paths.

Testing Libraries. We would ideally test a concurrent library over all inputs and calling contexts, but as this is often infeasible, we instead might want to *prioritize* the specific contexts a library is called from by a specific program. One such prioritization strategy is to enumerate all pairs of calls into the library that may run concurrently, then treat each pair as a program context that can be symbolically executed using our techniques. Then do the same for every triple of concurrent calls, every quadruple, and so on.

Piecewise Program Testing. Rather than testing a whole program with one long symbolic execution, we can break the program into adjacent fragments and test each fragment in isolation. Such a piecewise testing scheme might enumerate fragments *dynamically* by queuing the next fragments found to be reachable from the current fragment. Fragments might end at loop backedges, for loops with input-dependent iteration counts, producing a set of fragments that are each short and largely acyclic. The key potential advantage is that we can explore fragments in parallel, as they are enumerated, enabling us to more quickly reach a variety of deep paths in the program's execution. The trade-off we make is a potential loss of precision, as our dataflow analysis may make conservative assumptions when constructing a fragment's initial abstract state.

Execution Reconstruction. We can record an execution with periodic lightweight checkpoints that include call stacks and little else. Then, on a crash, we can symbolically execute from a checkpoint onwards to reconstruct the bug. Variants of this approach include `bbr` [29] and *RES* [101]. However, `bbr` does not work for multithreaded programs, and both systems have less powerful support for pointers than does our semantics.

Input-Covering Schedules. In Chapter 3, we show how the analysis described in this chapter can be used as a subroutine in an algorithm for enumerating input-covering schedules. That chapter includes a futher empirical evaluation of the symbolic execution techniques that we present here.

$$
\begin{array}{rll}
r \in \mathit{Var} & \textit{(local variables)} \\
x, y \in \mathit{SymbolicConst} & \textit{(symbolic constants)} \\
f \in \mathit{FName} & \textit{(function names)} \\
i \in \mathbb{Z} & \textit{(integers)}
\end{array}
$$

$$
\begin{array}{rcl}
v \in \mathit{Value} & ::= & f \mid i \\
e \in \mathit{Expr} & ::= & v \mid r \mid x \mid e \wedge e \mid e \vee e \mid e < e \mid \ldots
\end{array}
$$

$$
\begin{array}{rcl}
\gamma \in \mathit{StmtLabel} & & \\
s \in \mathit{Stmt} & ::= & r \leftarrow e(e^*) \\
& & \mid \texttt{br}\ e,\ \gamma_t,\ \gamma_f \\
& & \mid \texttt{return}\ e \\
\mathit{Func} & ::= & \texttt{func}\ f(r^*)\{\ (\gamma : s;)^*\ \}
\end{array}
$$

Figure 2.2: Syntax of *Simp*. Asterisks (*) denote repetition.

2.3 A Simple Imperative Language

Figure 2.2 gives the syntax of *Simp*, a simple imperative language that we use as a starting point. A program in this language contains a set of functions, including a distinguished `main` function for program entry. The language includes function calls, conditional branching, mutable local variables, and a set of standard arithmetic and boolean expressions (only partially shown). We separate side-effect-free expressions from statements. This simple language does *not* include pointers, dynamic memory allocation, or threads—those language features will be added in Sections 2.4 and 2.5, respectively.

The concrete semantics follow the standard pattern for imperative, lexically-scoped, call-by-value languages.[1] Note that we use r to refer to local variables (or "registers"), while the metavariables x and y do not appear in the actual concrete language. Instead, x and y are used to name *symbolic constants* that represent unknown values during symbolic execution, as described below. *Simp* is intentionally left untyped—this conservatively models our implementation (Section 2.8), which operates over a low-level language (LLVM bitcode [68]) that has very weak and limited types.

[1] A complete listing of the concrete semantics for the fully-featured core language, *SimpThreads*, is given in Appendix A. *Simp* is a strict subset of *SimpThreads*.

$\overline{\mathcal{Y}}$	:	Stack of $(Var \rightarrow Expr)$	*(local variables)*
CallCtx	:	Stack of *StmtLabel*	*(calling context)*
path	:	List of *StmtLabel*	*(execution trace)*
C	:	*Expr*	*(path constraint)*

Figure 2.3: Symbolic state for *Simp*.

Challenges. Although this language is simple, it reveals two ways in which symbolic execution from arbitrary contexts can be imprecise. Specifically, we use this language to demonstrate imprecision due to unknown calling contexts (Section 2.3.2) and unknown values of local variables (Section 2.3.3). We also use this language to present basic frameworks that we will reuse in the rest of this chapter.

2.3.1 Symbolic Semantics Overview

We now describe an analysis to perform symbolic execution of *Simp* programs. Our analysis operates over symbolic states that contain the domains illustrated in Figure 2.3:

- $\overline{\mathcal{Y}}$, which is a stack of local variable bindings. A new stack frame is pushed by each function call and popped by the matching return. Variables are bound to either function arguments, for formal parameters, or the result of a function call, as in the statement $r \leftarrow f()$.

- *CallCtx*, which names the current calling context, where the youngest stack label is the current program counter and older labels are return addresses. We use γ_{curr} to refer to the current program counter.

- *path*, which records an execution trace.

- C, an expression that records the current path constraint.

Constructing an Initial State. Recall from Section 2.1 that our job is to perform symbolic execution from an arbitrary program context that is specified by a set of program counters, one for each thread. As *Simp* is single-threaded, the initial program context for *Simp* programs contains just one program counter, γ_0.

Given γ_0, where γ_0 is a statement in function f_0, we must construct an initial symbolic state, S_{init}, from which we can begin symbolic execution. A simple approach is: $path_{init} = empty$; $C_{init} = \text{true}$; $CallCtx_{init} = \{\gamma_0\}$ (but see Section 2.3.2 for a caveat); and $\overline{\mathcal{Y}}_{init}$ contains one stack frame that maps each $r_i \in f_0$ to a distinct symbolic constant x_i. This simple approach is clearly correct, as it constructs an initial symbolic state that over-approximates all possible concrete initial states. However, this simple approach is very imprecise. We describe a more precise approach in Section 2.3.3.

Correspondence of Concrete and Symbolic States. Note that we use *symbolic constants*, such as x_i, above, to represent unknown parts of a symbolic state. This allows each symbolic state to represent a *set* of concrete states. Specifically, the set of concrete states represented by S_{init} can be found by enumerating the total set of assignments of symbolic constants x_i to values v_i—each such assignment corresponds to a concrete state in which $x_i = v_i$.

Symbolic Execution. At a high level, symbolic execution is straightforward. We begin from the initial state, S_{init}. We execute one statement at a time using *step*, which is defined below. At branches, we use an SMT solver to determine which branch edges are feasible and we fork as necessary. We repeatedly execute *step* on non-terminated states until all states have terminated or until a user-defined resource budget has been exceeded. We define *step* as follows, and we also make use of an auxiliary function *eval* to evaluate side-effect-free expressions:

- $step : State \rightarrow \text{Set of } State$

  Evaluates a single statement under an initial state and produces a set of states, as we may *fork* execution at control flow statements to separately evaluate each feasible branch. The type of each *State* is given by Figure 2.3. Each invocation of *step* evaluates the statement referenced by current program counter, γ_{curr}, then appends γ_{curr} to the *path* and advances the program counter.

$isSat(C, e)$ = true iff e is satisfiable under C
$mayBeTrue(C, e) = isSat(C, e)$
$mustBeTrue(C, e) = \neg mayBeTrue(C, \neg e)$

$read(A, e_{off}) = A(e_{off})$
$write(A, e_{off}, e_{value}) = A[e_{off} \mapsto e_{val}]$

Figure 2.4: Interface to the off-the-shelf SMT solver.

- $eval : ((Var \to Expr) \times Expr) \to Expr$

  Given $eval(\mathcal{Y}, e)$, we evaluate expression e under binding $\mathcal{Y}$, where $\mathcal{Y}$ represents a single stack frame. We expect that $\mathcal{Y}$ has a binding for every local variable referenced in expression e (non-existent bindings are a runtime error). Note that *eval* returns an *Expr* rather than a *Value*, as we cannot completely reduce expressions that contain symbolic constants.

The final result of our symbolic analysis is a set of *State*s from which we can extract (*path*, C) pairs that represent our final output. For each such pair, C is an expression that constrains the initial symbolic state, S_{init}, such that when C is satisfied, program execution *must* follow the corresponding *path*.

SMT Solver Interface. Our symbolic semantics relies on an SMT solver that must support, at minimum, basic integer arithmetic and the theory of arrays. We query that solver using the interface shown in Figure 2.4. The function $isSat(C, e)$ determines if there exists a binding from symbolic constants to values such that boolean expression e is satisfiable under the constraints given by expression C, where C is a conjunction of assumptions. In addition to *isSat*, we use *mayBeTrue* and *mustBeTrue* as syntactic sugar. The functions *read* and *write* are standard constructors from the theory of arrays (*e.g.*, see [46]).

If a query $isSat(C, e)$ cannot be solved, then our symbolic execution becomes *incomplete*. In this case, we concretize enough subexpressions of e so the query becomes solvable and we can make forward progress, similarly to Pasareanu *et al.* [83]. For example, suppose *isSat* cannot reason about modulo arithmetic, but we are given the expression $x\%y < z$: in this case, we select concrete values i_x and i_y for x and y such that $isSat(C, x = i_x \wedge y = i_y)$, where C is the current path constraint, and then we append $x = i_x \wedge y = i_y$ to the path constraint so that x and y are effectively concretized on all paths that follow.

2.3.2 Dealing with an Underspecified CallCtx

Recall that the initial program context is simply a single program counter, γ_0. If γ_0 is not a statement in the `main` function, then the initial state S_{init} does *not* have a complete call stack. How do we reconstruct a complete call stack?

We could start with a single stack frame and then lazily expand older frames, forking as necessary to explore all paths through the static call graph. However, we consider this overkill for our anticipated applications, and instead opt to exit the program when the initial stack frame returns. Our rationale is that, for each application listed in Section 2.2, either the program fragment of interest will be lexically scoped, in which case we never return from the initial stack frames anyway, or complete call stacks will be provided, which we can use directly (*e.g.*, we expect that complete call stacks will be available during execution reconstruction, as in `bbr` [29], and also during input-covering schedule enumeration, as described in Chapter 3).

2.3.3 Initializing Local Variables with Reaching Definitions

The simple approach for constructing S_{init}, as described above, is imprecise. Specifically, the simple approach assigns each local variable a unique symbolic constant, x_i, effectively assuming that each local variable can start with *any* initial value. This is often not the case. For example, consider thread t_1 in Figure 2.1. In this example, assuming that `RunA` is the only caller of `Foo`, the value of local variable `a` is known precisely. Even when the initial value of a variable cannot be determined precisely, we can often define its initial value as a symbolic function over other variables.

Our approach is to initialize $\overline{\mathcal{Y}}_{init}$ using an interprocedural dataflow analysis that computes *reaching definitions* for all local variables. We use a standard iterative dataflow analysis framework with function summaries for scalability, and we make the framework *context-specific* as follows: First, we combine a static call graph with each function's control-flow graph to produce an interprocedural control-flow graph, *CFG*. Then, we run a forwards dataflow analysis over *CFG* that starts from `main` and summarizes all interprocedural paths between program entry and the initial program counter, γ_0.

Our dataflow analysis computes assignments that *must-reach* the initial program context. Specifically, we compute a set of pairs $R_{local} = \{(r_i, e_i)\}$, where each r_i is a local variable in $\overline{\mathcal{Y}}_{init}$ such that the assignment $r_i \leftarrow e_i$ must-reach the statement γ_0. That is, for each pair $(r_i, e_i) \in R_{local}$, r_i's value at the initial program context *must* match expression e_i.

After computing R_{local} for the initial program context, we initialize $\overline{\mathcal{Y}}_{init}$ as follows: for each pair $(r_i, e_i) \in R_{local}$, we assign e_i to r_i in $\overline{\mathcal{Y}}_{init}$. Some variables may not have a must-reach assignment—these variables, r_k, do not appear in R_{local}, and they are assigned a unique symbolic constant x_k in $\overline{\mathcal{Y}}_{init}$, as before.

We save a formal description of our reaching definitions analysis for later in this chapter, in Section 2.6.2, at which point we formally describe how our reaching definitions analysis applies to the full *SimpThreads* language.

Must-Reach vs. May-Reach. Must-reach definitions provide a sound *over-approximation* of $\overline{\mathcal{Y}}_{init}$, as any variable not included in the must-reach set may have *any* initial value. More precision could be achieved through *may-reach* definitions; however, this would result in a symbolic state with many large disjunctions that are expensive to solve in current SMT solvers [52, 61].

2.4 Adding Pointers

Figure 2.5 shows the syntax of *SimpHeaps*, which adds pointers and dynamic memory allocation to *Simp*. As a convention, we use p to range over expressions that should evaluate to pointers.

Memory Interface. We represent pointers as pairs $\texttt{ptr}(l, i)$, where l is the base address of a heap object and i is a non-negative integer offset into that object. Pointers may also be `null`. Pointer arithmetic is supported with the $\texttt{ptradd}(p, e)$ expression, which is evaluated as follows in the concrete language:

$$\frac{eval(\mathcal{Y}, p) = \texttt{ptr}(l, i) \qquad eval(\mathcal{Y}, e) = i'}{eval(\mathcal{Y}, \texttt{ptradd}(p, e)) = \texttt{ptr}(l, i + i')}$$

The heap is a mapping from locations to objects, and each object includes a sequence

$$
\begin{array}{rll}
l \in \mathit{Loc} & & \textit{(heap locations)} \\
v \in \mathit{Value} & ::= & ... \mid \texttt{null} \mid \texttt{ptr}(l, i) \\
e, p \in \mathit{Expr} & ::= & ... \mid \texttt{ptr}(l, e) \mid \texttt{ptradd}(p, e) \\
s \in \mathit{Stmt} & ::= & ... \mid r \leftarrow \texttt{load}\ p \mid \texttt{store}\ p,\ e \\
& & \mid r \leftarrow \texttt{malloc}(e) \mid \texttt{free}(p)
\end{array}
$$

Figure 2.5: Syntax additions for *SimpHeaps*.

of fields. Our notion of a field encompasses the common notions of array elements and structure fields. To simplify the semantics, we assume that each field has a uniform size that is big enough to store any value. Following that assumption, we define i to be the offset of the (i+1)th field (making 0 the offset of the first field), and we define the size of an object to be its number of fields. Our implementation (Section 2.8) relaxes this assumption to support variable-sized fields at byte-granular offsets. Heap objects are allocated with `malloc`, which returns $\texttt{ptr}(l, 0)$ with a fresh location l, and they are deallocated with `free`.

Memory Errors. Out-of-bounds memory accesses, uninitialized memory reads, and other memory errors have undefined behavior in C [55]. We treat these as runtime errors in our semantics to simplify the notions of *soundness* and *completeness* of symbolic execution. The problem of constructing dynamic detectors for these errors is well researched and orthogonal to the novelties in this dissertation, so we do not address that problem in detail.

Challenges. In the concrete language, `load` and `store` statements always operate on values of the form $\texttt{ptr}(l, i)$. The symbolic semantics must consider three additional kinds of pointer expressions: $\texttt{ptr}(l, e)$, in which the offset e is symbolic; and x and $\texttt{ptradd}(x, e)$, in which the heap location is symbolic as well.

2.4.1 Symbolic Semantics

We now extend our symbolic execution analysis for *SimpHeaps*. As shown in Figure 2.6, we add two fields to the symbolic state: a heap, $\mathcal{H}$, which maps concrete locations to dynamically allocated heap objects, and a list $\mathcal{A}$, which tracks aliasing information that is used to resolve symbolic pointers. Rules for the semantics described in this section are given in Figures 2.7, 2.8, and 2.10. The core rules involve the $\overset{mem}{\Longrightarrow}$ relation, which is used by *step*

$$\begin{array}{lll} \mathcal{H} & : & \mathit{Loc} \to \{\text{fields} : (\mathit{Expr} \to \mathit{Expr}), \} \qquad \qquad \textit{(heap)} \\ \mathcal{A} & : & \text{List of } \{\text{x} : \mathit{SymbolicConst},\ \text{primary} : \mathit{Loc},\ \text{n} : \mathit{PtrNode}\} \qquad \textit{(aliasable objects)} \end{array}$$

Figure 2.6: Symbolic state additions for *SimpHeaps*.

$$\boxed{\begin{array}{l} \mathit{heapGet} : (\mathit{Heap} \times \mathit{PtrExpr}) \to \mathit{Expr} \\ \mathit{heapPut} : (\mathit{Heap} \times \mathit{PtrExpr} \times \mathit{Bool} \times \mathit{Expr}) \to \mathit{Heap} \end{array}}$$

$$\frac{\begin{array}{c}(l, \{\text{fields}\}) \in \mathcal{H} \\ \mathit{read}(\text{fields}, e_{\mathit{off}}) = e\end{array}}{\mathit{heapGet}(\mathcal{H}, \texttt{ptr}(l, e_{\mathit{off}})) = e} \qquad \frac{\begin{array}{c}(l, \{\text{fields}\}) \in \mathcal{H} \qquad \mathit{read}(\text{fields}, e_{\mathit{off}}) = e_{\mathit{old}} \\ e_{\mathit{val}} = e_{\mathit{cond}} \ ?\ e_{\mathit{new}} : e_{\mathit{old}} \\ \mathit{write}(\text{fields}, e_{\mathit{off}}, e_{\mathit{val}}) = \text{fields}' \qquad \mathcal{H}' = \mathcal{H}[l \mapsto \{\text{fields}'\}]\end{array}}{\mathit{heapPut}(\mathcal{H}, \texttt{ptr}(l, e_{\mathit{off}}), e_{\mathit{cond}}, e_{\mathit{new}}) = \mathcal{H}'}$$

Figure 2.7: Symbolic heap interface, including conditional *put*.

to evaluate memory statements. (Note that memory operations never fork execution in the absence of memory errors, and since we elide error-checking details from this dissertation, the $\overset{mem}{\Longrightarrow}$ relation does not need to fork.)

Accessing Concrete Locations. We first consider accessing pointers of the form $\texttt{ptr}(l, e)$. In this case, l uniquely names the heap object being accessed, so we simply construct an expression in the theory of arrays to load from or store to offset e of that object's `fields` array.

Accessing Symbolic Locations. Now we consider accessing pointers of the form x and $\texttt{ptradd}(x, e)$. This case is more challenging since the pointer x may refer to an unknown object. Following [29], our approach is to assign each symbolic pointer x a unique *primary object* in the heap, then use aliasing constraints to allow multiple pointers to refer to the same object. This effectively encodes multiple concrete memory graphs into a single symbolic heap. We allocate the primary object for x *lazily*, the first time x is accessed. In this way, we lazily expand the symbolic heap and are able to efficiently encode heaps with unboundedly many objects.

Stores to x update x's primary object, l_x, and also conditionally update all other objects that x may-alias. For example, suppose pointers x and y may point to the same object. To

$$\boxed{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \mathit{Stmt} \overset{mem}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}'; C'; \mathcal{A}'}$$

Load/store of a concrete location:

$$\frac{\begin{array}{c} \mathit{eval}(\mathcal{Y}, p) = \mathtt{ptr}(l, e_{\mathit{off}}) \\ \mathit{heapGet}(\mathcal{H}, \mathtt{ptr}(l, e_{\mathit{off}})) = e \end{array}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; r \leftarrow \mathtt{load}\ \ p \overset{mem}{\Longrightarrow} \mathcal{H}; \mathcal{Y}[r \mapsto e]; C; \mathcal{A}}$$

$$\frac{\begin{array}{c} \mathit{eval}(\mathcal{Y}, p) = \mathtt{ptr}(l, e_{\mathit{off}}) \qquad \mathit{eval}(\mathcal{Y}, e) = e' \\ \mathit{heapPut}(\mathcal{H}, \mathtt{ptr}(l, e_{\mathit{off}}), \text{true}, e') = \mathcal{H}' \end{array}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \mathtt{store}\ \ p,\ e \overset{mem}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}; C; \mathcal{A}}$$

Load/store of a symbolic location:

$$\frac{\begin{array}{c} \mathit{eval}(\mathcal{Y}, p) = \mathtt{ptradd}(x, e_{\mathit{off}}) \\ \mathit{addPrimary}(\mathcal{H}, C, \mathcal{A}, x) = (\mathcal{H}', C', \mathcal{A}', \mathtt{ptr}(l_x, x_{\mathit{off}})) \\ \mathit{heapGet}(\mathcal{H}', \mathtt{ptr}(l_x, x_{\mathit{off}} + e_{\mathit{off}})) = e \end{array}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; r \leftarrow \mathtt{load}\ \ p \overset{mem}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}[r \mapsto e]; C'; \mathcal{A}'}$$

$$\frac{\begin{array}{c} \mathit{eval}(\mathcal{Y}, p) = \mathtt{ptradd}(x, e_{\mathit{off}}) \quad \mathit{eval}(\mathcal{Y}, e) = e' \\ \mathit{addPrimary}(\mathcal{H}, C, \mathcal{A}, x) = (\mathcal{H}', C', \mathcal{A}', \mathtt{ptr}(l_x, x_{\mathit{off}})) \\ \mathit{lookupAliases}(\mathcal{A}', x) = \{l_1 ... l_n\} \\ \mathit{heapPut}(\mathcal{H}', \quad \mathtt{ptr}(l_x, x_{\mathit{off}} + e_{\mathit{off}}), \text{true}, \qquad e') = \mathcal{H}''_0 \\ \mathit{heapPut}(\mathcal{H}''_0, \quad \mathtt{ptr}(l_1, x_{\mathit{off}} + e_{\mathit{off}}), (x = \mathtt{ptr}(l_1, x_{\mathit{off}})), e') = \mathcal{H}''_1 \\ \cdots \\ \mathit{heapPut}(\mathcal{H}''_{n-1}, \mathtt{ptr}(l_n, x_{\mathit{off}} + e_{\mathit{off}}), (x = \mathtt{ptr}(l_n, x_{\mathit{off}})), e') = \mathcal{H}''_n \end{array}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \mathtt{store}\ \ p,\ e \overset{mem}{\Longrightarrow} \mathcal{H}''_n; \mathcal{Y}; C'; \mathcal{A}'}$$

Allocate and free:

$$\frac{l = \mathit{fresh\ loc} \qquad \mathcal{H}' = \mathcal{H}[l \mapsto \{\lambda i.\mathtt{undef}\}]}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; r \leftarrow \mathtt{malloc}(e_{\mathit{size}}) \overset{mem}{\Longrightarrow} \mathcal{H}'; \mathcal{Y}[r \mapsto \mathtt{ptr}(l, 0)]; C; \mathcal{A}}$$

$$\frac{\mathit{true}}{\mathcal{H}; \mathcal{Y}; C; \mathcal{A}; \mathtt{free}(p) \overset{mem}{\Longrightarrow} \mathcal{H}; \mathcal{Y}; C; \mathcal{A}}$$

Figure 2.8: The $\overset{mem}{\Longrightarrow}$ relation in the symbolic heap semantics. In these rules, $\mathcal{Y}$ refers to the current stack frame (namely, the youngest stack frame in $\overline{\mathcal{Y}}$).

write value v to pointer x, we first update l_x by writing v to address $\texttt{ptr}(l_x, x_{\mathit{off}})$, and we then update l_y by writing the expression $(x = \texttt{ptr}(l_y, x_{\mathit{off}}))$? $v : e_{old}$ to address $\texttt{ptr}(l_y, e)$, where e_{old} is the previous value in l_y (this makes the update *conditional*) and x_{off} is a symbolic offset that will be described shortly. Loads of x access $\texttt{ptr}(l_x, x_{\mathit{off}})$ directly—since stores update all aliases, it is unnecessary for loads to access aliases as well.

Figure 2.8 shows the detailed semantics for accessing symbolic pointers of the form $\texttt{ptradd}(x, e_{\mathit{off}})$ (we treat x as $\texttt{ptradd}(x, 0)$).

Restricting Aliasing with a Points-To Analysis. Recall that symbolic constants like x represent values that originate in our initial program context. That is, if x is a valid pointer, then x *must* point to some object that was allocated *before* our initial program context. In the worst case, this set of possible aliases includes all primary objects that have been previously allocated for other symbolic pointers (in addition to any primary objects that may be allocated in the future). This list of primary objects is recorded in $\mathcal{A}$ (Figure 2.6) and kept up-to-date by *addPrimary* (Figure 2.10).

In practice, we can narrow the set of aliases using a static points-to analysis. On the first access to x, we add the record $\{x, l_x, n_x\}$ to $\mathcal{A}$, where n_x is the representative node for x in the static points-to graph. The set of objects that x *may-alias* is found by enumerating all $\{y, l_y, n_y\} \in \mathcal{A}$ for which n_y and n_x may point-to the same object according to the static points-to graph—this search is performed by *lookupAliases* (Figure 2.10). Note that, in practice, the search for aliases can be implemented efficiently by exploiting the structure of the underlying points-to graph.

We use a *field-sensitive* points-to analysis so we can additionally constrain the offset being accessed. For each symbolic pointer x, we query the points-to analysis to compute a range of possible offsets for x, and then construct a fresh symbolic constant x_{off} that is constrained to that range. (This is the same x_{off} used above in the discussion of loads and stores.) For example, if x is known to point at a specific field, then x_{off} is fixed to that field. If a range of offsets cannot be soundly determined, x_{off} is left unconstrained.

Hence, the points-to analysis must support two kinds of queries, as shown in Figure 2.9. *getPtrInfo* looks up a pointer x in the static points-to graph: it returns a tuple $(n_x, x_{\mathit{off}}, C)$, where n_x is the representative node for x in the static points-to graph, x_{off}

$getPtrInfo : Var \to (PtrNode, Expr, Expr, Expr)$
$mayAlias : (PtrNode, PtrNode) \to Bool$
$staticCallTargets : Expr \to \text{Set of } FName$

Figure 2.9: Interface to the off-the-shelf static points-to analysis.

is a symbolic constant, and C constrains x_{off} to be a possible offset for x, as described above. $mayAlias(n_x, n_y)$ returns true iff the pointers n_x and n_y may alias. *staticCallTargets* returns a set of functions that may be pointed-to by a function pointer expression—later in this section, we describe how *staticCallTargets* is used to resolve function pointers.

Constructing a New Primary Object. On the first access of symbolic pointer x, *addPrimary* allocates a primary object at l_x and appends the record $\{x, l_x, n_x\}$ to $\mathcal{A}$. The fields of l_x must be initialized carefully.

Suppose the first access of x is a load, and suppose that x may-alias some other symbolic pointer y. For soundness, we *must* ensure that every load of x satisfies the following invariant: $x = y \implies \texttt{load}(x) = \texttt{load}(y)$. Making matters more complicated is the fact that we may have performed stores on y before our first access of x, and we must ensure that these stores are visible through x as well. Our approach is to define the initial fields of l_x as follows:

$$\begin{aligned} &\textit{Initial}\ \texttt{fields}\ \textit{of}\ l_x \\ &\equiv\ (x = \texttt{ptr}(l_y, x_{\mathit{off}}))\ ?\ \texttt{fields}_y :\ \mathit{fresh} \end{aligned} \tag{2.1}$$

where $\texttt{fields}_y$ is the current fields array of object l_y, which is the primary object for y, and where *fresh* is a symbolic array that maps each field $\mathit{fresh}(i)$ to a fresh symbolic constant—this represents the unknown initial values of l_x in the case that x and y do not alias. In general x may have more than one alias, in which case we initialize the `fields` of l_x similarly to the above, but we use a chain of conditionals that compares x with all possible aliases. This initialization is performed by *getInitialFields* (Figure 2.10).

Memory Allocation. Semantics for $\texttt{malloc}(e_{\mathit{size}})$ are shown in Figure 2.8. Since each object has its own symbolic `fields` array, we naturally support allocations of unbounded symbolic size.

$$\boxed{\mathit{getInitialFields} : (\mathit{Heap} \times \text{Set of } \mathit{Loc} \times \mathit{Var} \times \mathit{Expr}) \to (\mathit{Expr} \to \mathit{Expr})}$$

$$\frac{\begin{array}{c}\{(l_i, f_i) \mid l_i \in \mathit{aliases} \wedge \mathcal{H}(l_i) = \{f_i\}\} \\ f'_1 {=} (x = \texttt{ptr}(l_1, x_{\mathit{off}})\ ?\ f_1 : \mathit{fresh}) \\ f'_2 {=} (x = \texttt{ptr}(l_2, x_{\mathit{off}})\ ?\ f_2 : f'_1) \\ \cdots \\ f'_n {=} (x = \texttt{ptr}(l_2, x_{\mathit{off}})\ ?\ f_n : f'_{n-1})\end{array}}{\mathit{getInitialFields}(\mathcal{H}, \mathit{aliases}, x, x_{\mathit{off}}) = f'_n}$$

$$\boxed{\begin{array}{l}\mathit{addPrimary} : (\mathit{Heap} \times \mathit{Expr} \times \mathit{AliasableObjects} \times \mathit{Var}) \\ \qquad\qquad\quad \to (\mathit{Heap} \times \mathit{Expr} \times \mathit{AliasableObjects} \times \mathit{Expr})\end{array}}$$

$$\frac{\begin{array}{c}\{x, \_, \_\} \notin \mathcal{A} \qquad \mathit{getPtrInfo}(x) = (n, x_{\mathit{off}}, C') \qquad \mathcal{A}' = \{x, l_x, n\} \cdot \mathcal{A} \\ \mathit{lookupAliases}(\mathcal{A}, x) = \mathit{aliases} \qquad \mathit{getInitialFields}(\mathcal{H}, \mathit{aliases}, x, x_{\mathit{off}}) = \mathit{fields} \\ l_x \notin \mathcal{H} \qquad \mathcal{H}' = \mathcal{H}[l_x \mapsto \{\mathit{fields}\}] \\ C'' = C \wedge C' \wedge \mathit{heapInvariants}(x, l_x, x_{\mathit{off}}, \mathit{aliases})\end{array}}{\mathit{addPrimary}(\mathcal{H}, C, \mathcal{A}, x) = (\mathcal{H}', C'', \mathcal{A}', \texttt{ptr}(l_x, x_{\mathit{off}}))}$$

$$\frac{\{x, l_x, \_\} \in \mathcal{A}}{\mathit{addPrimary}(\mathcal{H}, C, \mathcal{A}, x) = (\mathcal{H}, C, \mathcal{A}, \texttt{ptr}(l_x, x_{\mathit{off}}))}$$

$$\boxed{\mathit{lookupAliases} : (\mathit{AliasableObjects} \times \mathit{Var}) \to \text{Set of } \mathit{Loc}}$$

$$\frac{\begin{array}{c}\{x, l_x, n_x\} \in \mathit{AliasMap} \\ \mathit{aliases} = \{l_i \mid \{\_, l_i, n_i\} \in \mathcal{A} \wedge l_i \neq l_x \wedge \mathit{mayAlias}(n_x, n_i)\}\end{array}}{\mathit{lookupAliases}(\mathcal{A}, x) = \mathit{aliases}}$$

Figure 2.10: Operations that manipulate the *aliasable objects* list. The function *heap-Invariants* accumulates invariants that constrain the initial state of the newly allocated primary object—the origin of these invariants is described later, in Section 2.4.2.

Memory Error Checkers. Since this dissertation elides memory error-checking details, we treat $\texttt{free}(p)$ as a no-op in Figure 2.8.

Briefly, to detect memory errors, we might add *size* and *isLive* attributes to each object in $\mathcal{H}$. On $\texttt{malloc}(e)$, we would set $\mathit{size} = e$ and $\mathit{isLive} =$ true. On $\texttt{free}(p)$, we would *conditionally* free all objects that p may-alias by conditionally setting $\mathit{isLive} =$ false in all aliases, much in the same way that $\texttt{store}(p, e)$ conditionally writes e to all aliases of p. Error checkers such as out-of-bounds and use-after-free would then ensure that, for each access at $\texttt{ptr}(l, e_{\mathit{off}})$, $0 \leq e_{\mathit{off}} < \mathcal{H}(l).\mathit{size}$ and $\mathcal{H}(l).\mathit{isLive} =$ true.

Compound Symbolic Pointer Expressions. Figure 2.8 shows rules for load and store statements where the pointer p evaluates to an expression of the form $\texttt{ptr}(l, e)$, x, or $\texttt{ptradd}(x, e)$, but the result of $\mathit{eval}(\mathcal{Y}, p)$ can also have the form $\mathit{read}(\mathit{fields}, e_{\mathit{off}})$. This form appears when a pointer is read from the heap, since all heap accesses use the theory of arrays.

The difficulty is that there may be multiple possible values at e_{off}. For example, if *fields* is $\mathit{write}(\mathit{write}(\_, 1, x), e'_{\mathit{off}}, x')$, then we cannot evaluate this address without first resolving the symbolic pointers x and x'. Further, the values written by *write* can contain conditional expressions due to the conditional store performed by *heapPut*. So, in general, the fields array might include a chain of calls as in the following: $\mathit{write}(\mathit{write}(\_, 1, x), e'_{\mathit{off}}, e'' \; ? \; x' : x'')$.

Our approach is to walk the call chain of *write*s to build *guarded expressions* that summarize the possible values at offset e_{off}. If the value stored by a *write* is a conditional expression, we also walk that conditional expression tree while computing the guarded expressions. This gives each guarded expression the form $e_{\mathit{grd}} \to p$, where each p has the form x, $\texttt{ptradd}(x, e)$, or $\texttt{ptr}(l, e)$. In the above example, we build guarded expressions $(e_{\mathit{off}} = e'_{\mathit{off}} \wedge e'') \to x'$, and $(e_{\mathit{off}} = e'_{\mathit{off}} \wedge \neg e'') \to x''$, and $(e_{\mathit{off}} \neq e'_{\mathit{off}} \wedge e_{\mathit{off}} = 1) \to x$, and so on down the call chain.

We then execute the memory operation on this set of guarded expressions. For stores, we evaluate each guarded expression independently: given $e_{\mathit{grd}} \to p$, we evaluate p using the rules in Figure 2.8, but we include e_{grd} in the condition passed to *heapPut*. For loads, we use the rules in Figure 2.8 to map each pair $e_{\mathit{grd}} \to p$ to a pair $e_{\mathit{grd}} \to e$, where e is the value loaded from pointer p. We then collect each $e_{\mathit{grd}} \to e$ into a conditional expression tree that represents the final value of the load. Continuing the above example, if the values at x, x', and x'' are v, v', and v'', respectively, then a load of the above example address would return the following conditional expression tree: $(e_{\mathit{off}} = e'_{\mathit{off}})\ ?\ (e''\ ?\ v' : v'') : (e_{\mathit{off}} = 1\ ?\ x : \_\_)$.

Function Pointers. At indirect calls, we first use *staticCallTargets* (recall Figure 2.9) to enumerate a set of functions F that might be called, according to the static points-to analysis. Then, we use *isSat* to prune functions from F that cannot be called given the current path constraint, and finally, we fork for each of the remaining possibilities.

2.4.2 Initializing the Heap with Reaching Definitions

The initial symbolic state (S_{init}) actually contains an *empty* heap that is expanded lazily, as described above. As the heap graph expands, newly uncovered objects are initially *unconstrained*, as represented by the *fresh* symbolic array allocated for each primary object (recall Equation (2.1), above). This approach can be imprecise for the same reasons discussed in Section 2.3.3. We improve precision using reaching definitions, as follows.

We extend the reaching definition analysis from Section 2.3.3 to also compute a set of heap writes that *must-reach* the initial program context. Specifically, we compute a set of pairs $R_{\mathit{heap}} = \{(p_i, e_i)\}$, where the heap location referenced by p_i *must* have a value matching e_i in the initial state. We use standard flow functions to compute R_{heap} and we use a static points-to analysis to reason about aliasing. A formal description of this analysis is given in Section 2.6.2.

We exploit R_{heap} during symbolic execution via *heap invariants* (recall Figure 2.10). Specifically, when adding a primary object l_x for symbolic pointer x, we invoke *heapInvariants* to construct an invariant on the initial values of object l_x. This invariant, shown below, is appended to the path constraint (again, recall Figure 2.10):

$$\bigwedge_{(\texttt{ptradd}(x, e_{\mathit{off}}), e_{\mathit{val}}) \in R_{\mathit{heap}}} \mathit{read}(\mathit{fresh}, x_{\mathit{off}} + e_{\mathit{off}}) = e_{\mathit{val}}$$

That is, we enumerate all pairs $(p, e_{\mathit{val}}) \in R_{\mathit{heap}}$ where p has either the form $\texttt{ptradd}(x, e_{\mathit{off}})$ or the form x (which we treat like $\texttt{ptradd}(x, 0)$). For each such pair (p, e_{val}), we constrain $\mathit{fresh}(x_{\mathit{off}} + e_{\mathit{off}}) = e_{\mathit{val}}$, where *fresh* is the initial symbolic array for l_x as shown in Equation

$$
\begin{aligned}
s \in \mathit{Stmt} ::= \; & ... \mid \texttt{threadCreate}(e_f, e_{arg}) \mid \texttt{yield}() \\
& \mid \texttt{wait}(p) \mid \texttt{notifyOne}(p) \mid \texttt{notifyAll}(p) \\
& \underline{\textit{synchronization annotations}} \\
& \mid \texttt{acquire}(p) \mid \texttt{release}(p) \\
& \mid \texttt{barrierInit}(p, e) \mid \texttt{barrierArrive}(p)
\end{aligned}
$$

Figure 2.11: New statements for *SimpThreads*.

(2.1) and Figure 2.10. Essentially, this invariant reflects the fact that the initial value of l_x at offset $x_{\mathit{off}}+e_{\mathit{off}}$ *must* have value e_{val}.

2.5 Adding Threads and Synchronization

Figure 2.11 shows the syntax additions for *SimpThreads*, which adds shared-memory multithreading and synchronization to *SimpHeaps*.

Threads. *SimpThreads* supports cooperative thread scheduling with `yield()`, which nondeterministically selects another thread to run. Cooperative scheduling with `yield` is sufficient to model any data race free program, as we can insert a `yield` at each synchronization operation to model all possible synchronization orderings. As with other memory errors (recall Section 2.4), data races have undefined behavior in C [17, 55] and are runtime errors in *SimpThreads*. Hence, cooperative scheduling is a valid model as we can assume that all *SimpThreads* programs are either data race free or will halt before the first race.

New threads are created by $\texttt{threadCreate}(e_f, e_{arg})$. This spawns a new thread that executes the function call $e_f(e_{arg})$, and the new thread will run until e_f returns. As *SimpThreads* uses cooperative scheduling, the new thread is *not* scheduled until another thread yields control.

Synchronization. We build higher-level synchronization objects such as barriers, condition variables, and queued locks using two primitives: cooperative scheduling with `yield`, which provides simple atomicity guarantees, and FIFO wait queues, which provide simple notify/wait operations that are common across a variety of synchronization patterns. Wait queues support three operations: `wait`, to yield control and move the current thread onto a wait queue; `notifyOne`, to wake the thread on the head of a wait queue; and `notifyAll`,

```
struct mutex {
  int taken;                          // at offset $i_{taken}$
}
pthread_mutex_init(mutex *m) {
  store ptradd(m, $i_{taken}$), 0;        // m->taken = 0
}
pthread_mutex_lock(mutex *m) {
  while (load ptradd(m, $i_{taken}$))     // while (m->taken)
    wait(ptradd(m, $i_{taken}$));         //   wait(&m->taken)
  store ptradd(m, $i_{taken}$), 1;        // m->taken = 1
  acquire(m);                         // acquire(m)
}
pthread_mutex_unlock(mutex *m) {
  store ptradd(m, $i_{taken}$), 0;        // m->taken = 0
  release(m);                         // release(m)
  notifyOne(ptradd(m, $i_{taken}$));      // notifyOne(&m->taken)
  yield();                            // yield()
}
```

Figure 2.12: Pseudocode demonstrating how pthreads' mutexes might be implemented in *SimpThreads*.

to wake all threads on a wait queue.

We use these building blocks to implement standard threading and synchronization libraries such as POSIX threads (pthreads). To aid our symbolic semantics, we assume synchronization libraries have been instrumented with the *annotation functions* listed in Figure 2.11. Annotation functions are no-ops that do not actually perform synchronization—they merely provide higher-level information that we will exploit, as described later (Section 2.5.3, Section 2.5.4). The example in Figure 2.12 demonstrates how to annotate an implementation of pthreads' mutexes. We have written the example in a pseudocode that uses memory operations resembling those in *SimpThreads*.

Note that wait queues are named by pointers. There is an implicit wait queue associated with every memory address—no initialization is necessary. For example, Figure 2.12 uses the implicit wait queue associated with `&m->taken`. This design is similar to both the `futex()` system call in Linux, which can be applied to any adress, and to `wait()` Java, which can be applied to any object. The reason for naming wait queues by an address rather than an integer id will become clear in Section 2.5.3.

Challenges. The primary challenge introduced by *SimpThreads* is the need to reason about synchronization objects. Our approach includes a semantics for symbolic wait queues (Section 2.5.2) and a collection of synchronization-specific invariants (Section 2.5.3) that exploit facts learned from a context-specific dataflow analysis (Section 2.5.4).

2.5.1 Symbolic Semantics

We now extend our symbolic execution semantics for *SimpThreads*. As illustrated in Figure 2.13, we modify $\overline{\mathcal{Y}}$ and *CallCtx* to include one call stack per thread, and we modify *path* to record a multithreaded trace. We add the following domains to the symbolic state:

- T^{Curr}, which is the id of the thread that is currently executing.
- T^{E}, which is the set of enabled threads, *i.e.*, the set of threads not blocked on synchronization. This includes T^{Curr}.
- WQ, which is a list that represents a global order of all waiting threads. Each entry of the list is a pair (p, t) signifying that thread t is blocked on the wait queue named by address p. The initial WQ can either be empty (all threads enabled) or non-empty (some threads blocked, as described in Section 2.5.2).
- $\mathtt{L}^{+}$, which describes a set of locks that *may* be held by each thread and is derived from `acquire` and `release` annotations.
- $\mathtt{B}^{\mathtt{cnts}}$, which describes a set of possible arrival counts for each barrier and is derived from `barrierInit` annotations.

$\mathtt{L}^{+}$ and $\mathtt{B}^{\mathtt{cnts}}$ are both *over-approximations*. They are initialized as described in Section 2.5.4 and they are used by invariants described in Section 2.5.3.

Symbolic Execution. Our first action during symbolic execution is to fork execution once for each possible $T^{Curr} \in T^{E}$. This effectively begins symbolic execution with an

$\overline{\mathcal{Y}}$	:	$\boldsymbol{\mathit{ThreadId}} \rightarrow$ Stack of $(\mathit{Var} \rightarrow \mathit{Expr})$	*(local variables)*
$\mathit{CallCtx}$	:	$\boldsymbol{\mathit{ThreadId}} \rightarrow$ Stack of $\mathit{StmtLabel}$	*(calling contexts)*
path	:	List of $(\boldsymbol{\mathit{ThreadId}}, \mathit{StmtLabel})$	*(execution trace)*

T^{Curr}	:	$\mathit{ThreadId}$	*(current thread)*
T^{E}	:	Set of $\mathit{ThreadId}$	*(enabled threads)*
WQ	:	List of $(\mathit{Expr}, \mathit{ThreadId})$	*(global wait queue)*
$\texttt{L}^{+}$	:	$\mathit{ThreadId} \rightarrow$ Set of Expr	*(acquired locksets)*
$\texttt{B}^{\texttt{cnts}}$	:	$\mathit{Expr} \rightarrow$ Set of Expr	*(barrier arrival cnts)*

Figure 2.13: Symbolic state modifications for *SimpThreads*, with modifications to *SimpHeaps* **bolded**, above the line, and additions shown below.

implicit `yield` so that each thread has a chance to run first. Note that context switches (updates to T^{Curr}) occur *only* either explicitly through `yield`, or implicitly when the current thread exits or is disabled through `wait`. Execution deadlocks when T^{E} is empty while WQ is non-empty, and execution terminates when both T^{E} and WQ are empty.

2.5.2 Symbolic Wait Queues

We now give symbolic semantics for the three FIFO wait queue operations, `wait`, `notifyOne`, and `notifyAll`. When a thread t calls `wait`(p), we remove t from T^{E} and append the pair (p, t) to WQ. When t is notified, we remove it from WQ and add it to T^{E}. *Which* threads are notified is answered as follows:

notifyOne(p). Any thread in WQ with a matching queue address may be notified. Let (p_1, t_1) be the first pair in WQ and let (p_n, t_n) be the last pair. We walk this ordered list and fork execution up to $|WQ| + 1$ times. The possible execution forks are given by the following list of path constraints:

$$\begin{aligned}
&(1) && p_1 = p \\
&(2) && p_1 \neq p \wedge p_2 = p \\
& && ... \\
&(\text{n}) && p_1 \neq p \wedge p_2 \neq p \wedge ... \wedge p_n = p \\
&(\text{n+1}) && p_1 \neq p \wedge p_2 \neq p \wedge ... \wedge p_n \neq p
\end{aligned}$$

In the first fork, we notify t_1, in the second, we notify t_2, and so on, until the nth fork, in which we notify t_n. In the final fork, no threads are notified. Only a subset of these forks may be feasible, so we use *isSat* to prune forked paths that have an infeasible path constraint. In particular, if there exists an i where $p_i{=}p$ *must* be true on the current path, then all forks from $(i{+}1)$ onwards are infeasible and will be discarded. Further, as in Section 2.4, we increase precision by using a static points-to analysis to determine when it *cannot* be true that $p_i{=}p$.

notifyAll(p). Any subset of threads in WQ may be notified. We first compute the powerset of WQ, $\mathcal{P}(WQ)$, and then fork execution once for each set $S \in \mathcal{P}(WQ)$. Specifically, on the path that is forked for set S, we notify all threads in S and apply the following path constraint:

$$\bigwedge_{(p_i,t_i)\in WQ} \begin{cases} p_i = p & \texttt{if } (p_i,t_i) \in S \\ p_i \neq p & \texttt{otherwise} \end{cases}$$

This forks execution $2^{|WQ|}$ ways. As before, we use *isSat* and a static points-to analysis to prune infeasible path constraints.

Initial Contexts with a Nonempty *WQ*. Suppose we want to analyze an initial program context in which some subset of threads begin in a *waiting* state, but we do not know the order in which the threads began waiting. One approach is to fork for each permutation of the wait order, but this is inefficient. Instead, our approach is to analyze such contexts by adding *timestamp counters*. First, we tag each waiting thread with a timestamp derived from a global counter that is incremented on every call to `wait`, so that thread t_1 precedes thread t_2 in WQ if and only if t_1's timestamp is less than t_2's timestamp.

Then, we set up the program context so that each waiting thread begins with the call to `wait` it is waiting in. Before beginning normal symbolic execution, we execute these `wait` calls in any order, using the semantics for `wait` described above, but with one adjustment: we give each waiting thread t_i a symbolic timestamp, represented by the symbol x_i, and we bound each $x_i < 0$ so these waits occur before other calls to `wait` during normal execution.

We say that $x_i < x_k$ is true in the concrete initial state when t_i and t_k are waiting on the same queue and t_i precedes t_k on that queue.

Next, we update the semantics of `notifyOne`. If there are n threads in WQ and w of those threads are *initial waiters*, meaning they have symbolic timestamps, then `notifyOne` uses the following sequence of path constraints, where $1 \leq i \leq w$:

$$\begin{array}{rl}
\text{(i)} & p_i = p \wedge \left(\bigwedge_{1 \leq k \leq w, k \neq i} (p_k = p) \Rightarrow (x_i < x_k) \right) \\
\text{(w+1)} & p_1 \neq p \wedge p_2 \neq p \wedge \ldots \wedge p_w \neq p \wedge p_{w+1} = p \\
 & \ldots \\
\text{(n)} & p_1 \neq p \wedge p_2 \neq p \wedge \ldots \wedge p_n = p \\
\text{(n+1)} & p_1 \neq p \wedge p_2 \neq p \wedge \ldots \wedge p_n \neq p
\end{array}$$

The first w constraints handle the cases where an initial waiter is notified. We can notify initial waiter t_i if it has a matching queue address, $p_i{=}p$, and it precedes all other initial waiters t_k with a matching address. The cases for w+1 and above are as before.

2.5.3 Synchronization Invariants

The semantics described above are sound, but the presence of unconstrained symbolic constants can cause our symbolic execution to explore infeasible paths. In an attempt to avoid infeasible paths, we augment the path constraint with higher-level program invariants.

Specifically, this section proposes a particularly high-value set of *synchronization invariants*. We focus on synchronization invariants here since the novelty of our work is symbolic exploration of multithreaded programs with symbolic synchronization. More generally, high-level invariants always help reduce explosion of infeasible paths and we could easily integrate programmer-specified invariants as well.

We cannot apply synchronization invariants without first identifying synchronization objects. Ideally we would locate such objects by scanning the heap, but our language *SimpThreads* is untyped, so we cannot soundly determine the type of an object by looking at it. Instead, we apply invariants when synchronization functions are called. For example, we instrument the implementation of `pthread_mutex_lock(m)` to apply invariants to `m` as the first step before locking the mutex. The rest of this section describes the invariants we have found most useful.

Locks. As illustrated in Figure 2.12, locks can be modeled by an object with a `taken` field that is non-zero when the lock is held and zero when the lock is released. Suppose a thread attempts to acquire a lock whose `taken` field is symbolic: execution must fork into two paths, one in which `taken`=0, so the lock can be acquired, and another in which `taken`≠`0`, so the thread must wait. One of these paths may be infeasible, as illustrated by Figure 2.1, so we need to further constrain lock objects to avoid such infeasible paths.

We use *locksets* to constrain the `taken` field of a lock object. Given a symbolic state with locksets $\mathtt{L}^+$ and a pointer p to some lock object, the lock's `taken` field can be non-zero only when there exists a thread T and an expression e, where $e \in \mathtt{L}^+(T)$, such that $e = p$. This invariant is expressed by the following constraint, where e_i ranges over all locks held by all threads:

$$(\mathtt{taken} = 0) \;\Leftrightarrow\; \left(\bigwedge_{e_i \in \mathtt{L}^+(*)} e_i \neq p \right)$$

Our dataflow analysis computes $\mathtt{L}^+$ for the initial symbolic state (Section 2.5.4). We keep $\mathtt{L}^+$ up-to-date during symbolic execution using the `acquire` and `release` annotations: on `acquire`(p) we add p to $\mathtt{L}^+(T^{Curr})$, and on `release`(p) we remove e from $\mathtt{L}^+(T^{Curr})$ where e must-equal p on the current path.

Barriers. A pthreads barrier can be modeled by two fields, `expected` and `arrived`, and a wait queue, where `arrived` is the number of threads that have arrived at the barrier, the barrier triggers when `arrived==expected`, and the wait queue is used to release threads when the barrier triggers.

Suppose a program has N threads spin in a loop, where each loop iteration includes a barrier with `expected=`N. Now suppose we analyze the program from an initial context where the barrier is unconstrained. When the first thread arrives at the barrier, execution forks at the condition `arrived==expected`. In the true branch we set `arrived=0` and notify the queue, and in the false branch we increment `arrived` and wait. This repeats for the other threads, and an execution tree unfolds in which we explore $\mathrm{O}(2^{\mathrm{N}})$ paths through a code fragment that has exactly one feasible path.

We compute invariants for both of these fields. Bounds for `arrived` can be determined by examining WQ: the number of threads that have arrived at a barrier is exactly the number of threads that are waiting on the barrier's wait queue. Let q be the wait queue address used by the barrier and let C be the current path constraint. We compute conservative lower- and upper-bounds for `arrived`. The lower-bound L is the number of pairs $(p,t) \in WQ$ for which $\mathit{mustBeTrue}(C, p{=}q)$, and the upper-bound H is the number of pairs for which $\mathit{mayBeTrue}(C, p{=}q)$. Given these bounds, the invariant is $L \leq \texttt{arrived} \leq H$.

A barrier's `expected` count is specified during barrier initialization. In pthreads, this occurs when `pthread_barrier_init` is called. Each symbolic state contains a $\texttt{B}^{\texttt{cnts}}$ that maps barrier pointers p to a set of expressions that describes the set of possible `expected` counts for all barriers pointed-to by p. So, we can use $\texttt{B}^{\texttt{cnts}}$ directly to construct an invariant for `expected`:

$$\bigwedge_{p' \in \texttt{B}^{\texttt{cnts}}} \left(\bigvee_{e \in \texttt{B}^{\texttt{cnts}}(p')} (p' = p) \Rightarrow (\texttt{expected} = e) \right)$$

The initial $\texttt{B}^{\texttt{cnts}}$ is computed by our dataflow analysis (Section 2.5.4) and it does not change during symbolic execution—when `pthread_barrier_init` is called during symbolic execution we write to the barrier's `expected` field directly, making $\texttt{B}^{\texttt{cnts}}$ irrelevant for this case.

Other Types of Synchronization. The invariant described above for a barrier's `arrived` field is more generally stated as an invariant on the size of a given wait queue, making it applicable to other data structures that use wait queues, such as condition variables and queued locks.

Why Wait Queues are Named by Address. For standard synchronization objects such as barriers, condition variables, and queued locks, different objects do not share the same wait queue. For example, notifying the queue of lock L should not notify threads waiting at any other lock. By using the address of L to name L's wait queue, we state this invariant implicitly.

For contrast, suppose we instead named wait queues by an integer id. We would be forced to add a `queueId` field to each lock, then state the following invariant: $\forall p_1, p_2 : (p_1 = p_2) \Leftrightarrow (id_1 = id_2)$, where p_1 and p_2 range over the set of pointers to locks, and where id_1 and id_2 are the `queueId` fields in p_1 and p_2, respectively. Stating this as an axiom would require enumerating the complete set of pointers to locks, which can be extremely inefficient.

2.5.4 Approximating the Initial State of Synchronization

We update the context-specific dataflow framework introduced in Section 2.3.3 to support multiple threads. Specifically, we apply the dataflow framework as described in Section 2.3.3 to each thread, separately, and then combine the per-thread results to produce a multithreaded analysis. We perform the following analyses for *SimpThreads*:

Reaching Definitions. We update the reaching definitions analysis described in Section 2.4.2 to support multiple threads. Importantly, since we analyze each thread in isolation, we must reason about cross-thread interference. Our approach is to label memory locations in R_{heap} as either *conflict-free* or *shared.* A location is conflict-free if it is provably thread-local (via an escape analysis) or if all writes to the location must-occur before the first call to `threadCreate`—the second case captures a common idiom where the main thread initializes global data that is kept read-only during parallel execution. Shared locations may have conflicts—we reason about these conflicts using *interference-free regions* [41], as Section 2.6.2 describes in more detail.

Locksets. We use a lockset analysis to compute $\mathtt{L}^+(T)$, the set of locks that *may* be held at thread T's initial program counter. Our analysis uses relative locksets as in Relay [94]: each function summary includes two sets, $\mathtt{L}^+_f$ and $\mathtt{L}^-_f$, where $\mathtt{L}^+_f$ is the set of locks that function f *may acquire* without releasing, and $\mathtt{L}^-_f$ is the set of locks that f *always releases* without first acquiring.

The key difference between our implementation and Relay's is that we compute may-be-held sets while Relay computes must-be-held sets. This reflects differing motivations: as a static race detector, Relay wants to know which locks *must* be held to determine if accesses are properly guarded, but we want to know which locks *may* be held to determine when two `lock()` calls may need to be serialized (as motivated by Figure 2.1). Hence, our $\mathtt{L}^+$ and $\mathtt{L}^-$ are may-acquire and must-release, while those used by Relay are must-acquire and may-release.

Barrier Expected Arrivals. To compute $\mathtt{B}^{\mathtt{cnts}}$, we simply enumerate all calls to `barrierInit`(p, e) that might be performed on some path from program entry up to the initial context, and for each such call, we add e to the set $\mathtt{B}^{\mathtt{cnts}}(p)$. This can be viewed as *may-reach* analysis applied to each barrier's `expected` field.

Barrier Matching. A large class of data-parallel algorithms use barriers to execute threads in lock-step. For example, a program might execute the following loop in N different threads, where each iteration happens in lock-step:

```
for (i=0; i < Z; ++i) { barrierArrive(b); ... }
```

Suppose we are given an initial program context in which each thread begins inside this loop. In this case, since the loop runs in lock-step, we know that all threads must start from the same dynamic loop iteration, so we can add a constraint that equates the loop induction variable, `i`, across all threads. This constraint is included in the initial path constraint, $S_{init}.C$.

This is the *barrier matching* problem: given two threads, must they pass the same sequence of barriers from program entry up to the initial context? Solutions have been proposed—we adapt [102], which builds *barrier expressions* to describe the possible sequence of barriers each thread might pass through. Two threads are barrier-synchronized if their barrier expressions are compatible.

The algorithm in [102] does not support our use case directly because it cannot reason about loops with input-dependent trip counts. So, we extend that algorithm by computing a symbolic trip count for each loop node in a barrier expression. Two loops match if their symbolic trips counts *must* be equal. We compute trip counts using a standard algorithm, but we discard trip counts that depend on *shared* memory locations (recall the definition of *shared*, from above). To determine if the trip count can be kept, we compute a backwards slice of the trip count expression and ensure that slice does not depend on any *shared* locations.

2.6 The Big Picture

The full syntax of *SimpThreads* is given in Figure 2.14. This combines the syntaxes presented previously in Figures 2.2, 2.5, and 2.11. We assume a standard set of arithmetic expressions, in addition to what is illustrated explicitly in Figure 2.14.

The rest of this section provides a cohesive and formal description of a few semantics details that prior sections explained in prose. Section 2.6.1 formally describes our high-level algorithm for symbolic execution, and Section 2.6.2 formally specifies our context-specific reaching definitions analysis.

2.6.1 Symbolic Semantics

Figure 2.15 gives pseudocode for our top-level symbolic execution algorithm. We start with an *InitialState* computed by our context-sensitive dataflow analysis. Given this initial state, our first action is to execute an implicit `yield()` to allow each thread to run first. We then repeatedly invoke *step* until all states have exited or deadlocked. Our final output is a set of symbolic states, *Final*, where the paths explored by those states are given by $S.path$ and $S.C$ for each $S \in \mathit{Final}$. Pseudocode for the *step* function is shown in Figure 2.16. Our pseudocode for *step* uses a few shorthands:

We use `append`(L, k) to append item k to list L. We use `youngest`(s) to extract the youngest (or "top-most") element from stack s. We use `currStmt`(S) to extract the current statement from symbolic state S, where the current statement is named by the label `youngest`$(S.\mathit{CallCtx}(S.T^{\mathit{Curr}}))$.

`setCurrentPC`$(\mathit{CallCtx}, T, \gamma)$, `pushPC`$(\mathit{CallCtx}, T, \gamma)$, and `popPC`$(\mathit{CallCtx}, T)$ return an updated copy of $\mathit{CallCtx}$: `setCurrentPC` changes the youngest label of stack $\mathit{CallCtx}(T)$ to γ, `pushPC` pushes γ onto the stack $\mathit{CallCtx}(T)$, and `popPC` pops the youngest frame from $\mathit{CallCtx}(T)$. `pushFrame`$(\overline{\mathcal{Y}}, T, f, \bar{e})$ and `popFrame`$(\overline{\mathcal{Y}}, T)$ return an updated copy of $\overline{\mathcal{Y}}$: `pushFrame` pushes a new stack frame for function f onto $\overline{\mathcal{Y}}(T)$, where f's formal parameters are bound to the values $\bar{e}$, and `popFrame` pops the youngest frame off of $\overline{\mathcal{Y}}(T)$.

$$
\begin{array}{ll}
r \in \mathit{Var} & \textit{(local variables)} \\
x, y \in \mathit{SymbolicConst} & \textit{(symbolic constants)} \\
f \in \mathit{FName} & \textit{(function names)} \\
l \in \mathit{Loc} & \textit{(heap locations)} \\
i \in \mathbb{Z} & \textit{(integers)}
\end{array}
$$

$$
\begin{array}{rcl}
v \in \mathit{Value} & ::= & f \mid i \mid \texttt{null} \mid \texttt{ptr}(l, i) \\
e \in \mathit{Expr} & ::= & v \mid r \mid x \mid e \wedge e \mid e \vee e \mid e < e \mid ... \\
 & & \mid \texttt{ptr}(l, e) \mid \texttt{ptradd}(p, e) \\
\gamma \in \mathit{StmtLabel} & & \\
s \in \mathit{Stmt} & ::= & \texttt{return}\ e \\
 & & \mid \texttt{br}\ e,\ \gamma_t,\ \gamma_f \\
 & & \mid r \leftarrow e(e^*) \\
 & & \mid r \leftarrow \texttt{load}\ p \mid \texttt{store}\ p,\ e \\
 & & \mid r \leftarrow \texttt{malloc}(e) \mid \texttt{free}(p) \\
 & & \mid \texttt{threadCreate}(e_f, e_{arg}) \mid \texttt{yield}() \\
 & & \mid \texttt{wait}(p) \mid \texttt{notifyOne}(p) \mid \texttt{notifyAll}(p) \\
 & & \underline{\textit{synchronization annotations}} \\
 & & \mid \texttt{acquire}(p) \mid \texttt{release}(p) \\
 & & \mid \texttt{barrierInit}(p, e) \mid \texttt{barrierArrive}(p) \\
\mathit{Func} & ::= & \texttt{func}\ f(r^*)\{\ (\gamma : s;)^*\ \}
\end{array}
$$

Figure 2.14: Full syntax of *SimpThreads*

Figure 2.15 Symbolic execution algorithm.

```
**Input**: An $\mathit{InitialState}$
**Output**: A set of states $\mathit{Final}$
**Data**: A worklist $w$
$w \leftarrow \{\ \mathit{InitialState}\ \textbf{with}\ \{T^{Curr} \leftarrow T\} \mid T \in \mathit{InitialState}.T^E\ \}$
$\mathit{Final} \leftarrow \emptyset$
while $w \neq \emptyset$ do
  $S \leftarrow \mathit{pickNext}(w)$
  if $S.T^E = \emptyset$ then
    $\mathit{Final} \leftarrow \mathit{Final} \cup \{S\}$
  else
    $w \leftarrow (w/S) \cup \mathit{step}(S)$
return $\mathit{Final}$
```

Figure 2.16 The *step* function. We write $S.X$ to extract domain X from state S, and we write S **with** $\{X \leftarrow ...\}$ to perform a functional update of domain X in state S. We use **yield** (as in Python) to return multiple forked states. Not yielding anything is equivalent to returning an empty set of states (*e.g.*, when the program exits or deadlocks). We use $S.\mathcal{Y}$ as shorthand for the current thread's youngest stack frame (*i.e.*, $\mathit{youngest}(S.\overline{\mathcal{Y}}(S.T^{Curr}))$).

Input: A state S
Output: A set of new states that result from evaluating S

```
match currStmt($S$):
  case br $e, \gamma_t, \gamma_f$:
    $e' \leftarrow \mathit{eval}(S.\mathcal{Y}, e)$
    if $\mathit{isSat}(S.C, e')$ then
      yield $S$ with { $\mathit{CallCtx}$  $\leftarrow$ setCurrentPC($S.\mathit{CallCtx}$, $S.T^{Curr}$, $\gamma_t$)
                     $\mathit{path}$     $\leftarrow$ append($S.\mathit{path}$, $(S.T^{Curr}, \gamma_t)$)
                     $C$        $\leftarrow S.C \wedge e'$ }
    if $\mathit{isSat}(S.C, \neg e')$ then
      yield $S$ with { $\mathit{CallCtx}$  $\leftarrow$ setCurrentPC($S.\mathit{CallCtx}$, $S.T^{Curr}$, $\gamma_f$)
                     $\mathit{path}$     $\leftarrow$ append($S.\mathit{path}$, $(S.T^{Curr}, \gamma_f)$)
                     $C$        $\leftarrow S.C \wedge \neg e'$ }
  case return $e$:
    $e' \leftarrow \mathit{eval}(S.\mathcal{Y}, e)$
    $\mathit{newCallCtx} \leftarrow$ popPC($S.\mathit{CallCtx}, S.T^{Curr}$)
    if $\mathit{newCallCtx}(S.T^{Curr}) = \{\}$ then          // did thread $S.T^{Curr}$ terminate?
      $T^E_{new} \leftarrow S.T^E \;/\; \{S.T^{Curr}\}$
      foreach $T \in T^E_{new}$ do
        yield $S$ with { $T^{Curr}$    $\leftarrow T$
                       $T^E$       $\leftarrow T^E_{new}$
                       $\overline{\mathcal{Y}}$        $\leftarrow S.\overline{\mathcal{Y}} \;/\; \{S.T^{Curr}\}$
                       $\mathit{CallCtx}$  $\leftarrow S.\mathit{CallCtx} \;/\; \{S.T^{Curr}\}$
                       $\mathit{path}$     $\leftarrow$ append($S.\mathit{path}$, ($T$, youngest($S.\mathit{CallCtx}(T)$))) }
    else
      yield $S$ with { $\overline{\mathcal{Y}}$        $\leftarrow$ popFrame($S.\overline{\mathcal{Y}}, S.T^{Curr}$)
                     $\mathit{CallCtx}$  $\leftarrow \mathit{newCallCtx}$
                     $\mathit{path}$     $\leftarrow$ append($S.\mathit{path}$, ($S.T^{Curr}$, youngest($\mathit{newCallCtx}$))) }
  ... (continued)
```

Figure 2.16 The *step* function (continued.)

```
  ...
  case $x \leftarrow e(\overline{e_1})$:
    $e' \leftarrow eval(S.\mathcal{Y}, e)$
    $\overline{e_1}' \leftarrow eval(S.\mathcal{Y}, \overline{e_1})$
    foreach $f \in staticCallTargets(e')$ do
      if $isSat(S.C,\ e'{=}f)$ then
        $\gamma_0 \leftarrow \text{entryStmt}(f)$
        yield $S$ with { $\overline{\mathcal{Y}}$      $\leftarrow \text{pushFrame}(S.\overline{\mathcal{Y}},\ T,\ f,\ \overline{e_1}')$
                         $CallCtx$  $\leftarrow \text{pushPC}(S.CallCtx,\ S.T^{Curr},\ \gamma_0)$
                         $path$     $\leftarrow \text{append}(S.path,\ (S.T^{Curr}, \gamma_0))$
                         $C$        $\leftarrow S.C \wedge e' = f$ }
  case threadCreate($e_f, e_{arg}$):
    $e'_f \leftarrow eval(S.\mathcal{Y}, e_f)$
    $e'_{arg} \leftarrow eval(S.\mathcal{Y}, e_{arg})$
    $T \leftarrow$ fresh thread id
    foreach $f \in staticCallTargets(e'_f)$ do
      $\gamma_0 \leftarrow \text{entryStmt}(f)$
      $\gamma_{next} \leftarrow \text{youngest}(S.CallCtx(S.T^{Curr})) + 1$
      yield $S$ with { $T^E$       $\leftarrow S.T^E \cup \{T\}$
                       $\overline{\mathcal{Y}}$         $\leftarrow \text{pushFrame}(S.\overline{\mathcal{Y}},\ T,\ f,\ e'_{arg})$
                       $CallCtx$   $\leftarrow \text{setCurrentPC}(\text{pushPC}(S.CallCtx,\ T,\ \gamma_0),\ S.T^{Curr},\ \gamma_{next})$
                       $path$      $\leftarrow \text{append}(S.path,\ (S.T^{Curr}, \gamma_{next}))$
                       $C$         $\leftarrow S.C \wedge e'_f = f$ }
  case yield():
    foreach $T \in S.T^E$ do
      $\gamma_{next} \leftarrow \text{youngest}(S.CallCtx(S.T^{Curr})) + 1$
      yield $S$ with { $T^{Curr}$   $\leftarrow T$
                       $CallCtx$   $\leftarrow \text{setCurrentPC}(S.CallCtx,\ S.T^{Curr},\ \gamma_{next})$
                       $path$      $\leftarrow \text{append}(S.path,\ (S.T^{Curr}, \gamma_{next}))$ }
  case wait($p$) | notifyOne($p$) | notifyAll($p$):
    evaluate as described in Section 2.5.2
  case acquire($p$) | release($p$) | barrierInit($p$) | barrierArrive($p$):
    evaluate as described in Section 2.5.3
  case $r \leftarrow$ load $p$ | store $p, e$ | $r \leftarrow$ malloc($e$) | free($p$):
    yield the invocation of $\overset{mem}{\Longrightarrow}$ on state $S$ (recall Section 2.4.1)
end
```

2.6.2 Context-Specific Dataflow Analysis for Reaching Definitions

Recall that for each thread T, where T's initial program counter is γ_0^T, we compute reaching definitions that summarize all paths from program entry up to γ_0^T. We use a standard forward, iterative dataflow analysis that is applied to an interprocedural control-flow graph, *CFG*. Reaching definitions is a standard and well-known analysis, so many of the details presented here should be familiar. However, two aspects of our analysis are non-standard: our definition of the interprocedural *CFG*, and our simultaneous treatment of the stack and the heap. We discuss both aspects below.

Constructing *CFG*. The interprocedural control-flow graph, *CFG*, is formed by merging all function-local control-flow graphs into a single graph, where the function-local subgraphs are connected using edges from the program's static call graph, as follows: At each node in *CFG* that represents an function call, we query a static points-to analysis to enumerate all functions that *might* be invoked by that call site. For each such target function f, we add a few edges to *CFG*: one edge from the call site to f, and another edge from each of f's return statements back to the call site. Hence, function calls implicitly invoke a branch (which models the function call) followed by a control-flow merge (which models the return from multiple potential call targets).

We also include special control-flow edges at `threadCreate`(e_f, e_{arg}) statements as follows: Suppose that e_f resolves to a set of functions F. Then, for each $f \in F$, then we add a control-flow edge from `threadCreate`(e_f, e_{arg}) to f. This allows our analysis to connect expression e_{arg} with the argument of function f, and further, it allows our analysis to reason about values passed on the heap from a creator thread to its child threads. However, we do *not* add a control-flow edge from f's return statements back to the `threadCreate` call, as there is no control-flow in this direction. Note that when the program counter γ_0^T is not reachable from function f, the control-flow edge connecting `threadCreate` to f is essentially "dead" and will not affect any dataflow facts computed for statement γ_0^T.

Running the Analysis. Our reaching definitions analysis computes R_{local} and R_{heap} for all statements in the program that are reachable from the entry point of *CFG*, where we define the entry point of *CFG* to be the first statement of the `main` function. After reaching definitions have been computed, we extract R^T_{local} and R^T_{heap} for each thread T, where we define R^T_{local} and R^T_{heap} to be the values of R_{local} and R_{heap} at statement γ^T_0. Essentially, R^T_{local} and R^T_{heap} represent thread T's contribution to the initial symbolic state.

We use R^T_{local} to initialize $\overline{\mathcal{Y}}_{init}(T)$ as described in Section 2.3.3. Namely, each local variable $r_k \in \overline{\mathcal{Y}}_{init}(T)$ is assigned the value $R^T_{local}(r_k)$ if $r_k \in R^T_{local}$, and otherwise, r_k is assigned a unique symbolic constant x_k.[2]

We union all R^T_{heap} into a global R^G_{heap}, where R^G_{heap} is the set of reaching definitions used by *heapInvariants* during symbolic execution (recall Section 2.4.2). Note that, because of how we reason about cross-thread interference (see below), a pointer p can exist in both R^{T1}_{heap} and R^{T2}_{heap} only if there is a race on location p—in this case, we discard p from R^G_{heap}.

Flow Functions. Figure 2.17 gives flow functions that define R_{local} and R_{heap} for each node in *CFG*. There are four cases. The first case, for generic assignments, is applied at function call statements such as $r \leftarrow e(\bar{e})$: once to assign the values $\bar{e}$ to the formal parameters of the callee, and a second time to assign the return value to local variable r. As we compute the outgoing R^{out}_{local}, each assignment of e to r generates a must-reach definition $R^{out}_{local}(r) = eval(R^{in}_{local}, e)$. Note that we use *eval* to reduce expressions. Thus, given $e = r_1{+}5$, where $R^{in}_{local}(r_1) = x$, we generate the definition $R^{out}_{local}(r) = x{+}5$ to express that r and r_1 are functions of the same value.

The second case deals with stores to the heap. Here, as is standard, we first kill all definitions in R^{in}_{heap} for locations that may-alias the pointer p. We then add a new definition to R^{out}_{heap}. As for assignments, we use *eval* to reduce this new definition.

The third case deals with a load from the heap at location p. Here, we check if there exists a heap assignment $R^{in}_{heap}(p_1) = e_1$ heap such that p_1 must-alias[3] the pointer p. If such an assignment is found, then e_1 is used as the value loaded. Otherwise, the value loaded is unknown, so we assign local variable r a uniquely chosen symbolic constant, x_r. We then memoize the assignment (p, x_r) in R^{out}_{heap} to implement a form of redundant load elimination:

[2] As *CFG* is interprocedural, R_{local} can collect reaching definitions from multiple functions. We implicitly assume (without loss of generality) that local variable names are not reused across functions, allowing r_k to name a specific variable in R_{local}.

[3] We implement *mustAlias*(p_1, p_2) by checking for syntactic equality of the expressions. For example, the expression $\texttt{ptr}(x, 5)$ is syntactically equivalent to $\texttt{ptr}(x, 5)$, but not to $\texttt{ptr}(x, 0)$ or $\texttt{ptr}(y, 5)$.

```
$\textit{Flow Functions for } R_{local} \textit{ and } R_{heap}$
  case $\textit{assign e to r}$:
    $R^{out}_{local} = R^{in}_{local}[r \mapsto eval(R^{in}_{local}, e)]$
  case store $p$, $e$:
    $killset = \{p_i \mid p_i \in R^{in}_{heap} \wedge mayAlias(p_i, p)\}$
    $R^{out}_{heap} = (R^{in}_{heap}/killset)[eval(R^{in}_{local}, p) \mapsto eval(R^{in}_{local}, e)]$
  case $r \leftarrow$ load $p$:
    $p' = eval(R^{in}_{local}, p)$
    if $\exists p_1$ s.t. $p_1 \in R^{in}_{heap} \wedge mustAlias(p_1, p')$:
      $R^{out}_{local} = R^{in}_{local}[r \mapsto R^{in}_{heap}(p_1)]$
    else:
      $x_r = uniqueSymbolicConstantFor(r)$
      $R^{out}_{local} = R^{in}_{local}[r \mapsto x_r]$
      $R^{out}_{heap} = R^{in}_{heap}[p' \mapsto x_r]$
  case $\textit{merge } br_1, br_2, \cdots br_n$:
    $R^{out}_{local} = R^1_{local} \cap R^2_{local} \cap \cdots \cap R^n_{local}$
    $R^{out}_{heap} = R^1_{heap} \cap R^2_{heap} \cap \cdots \cap R^n_{heap}$
```

Figure 2.17: Flow function for reaching definitions. This function is applied to each node in the interprocedural control-flow graph, *CFG*. The inputs to this function are R^{in} (or $R^1 \cdots R^n$ for merge nodes), and the outputs are R^{out}. Inputs come from incoming control-flow edges, while outputs are emitted onto each outgoing edge. For the entry node of *CFG*, R^{in}_{local} and R^{in}_{heap} are empty.

For example, if a later statement loads from the same pointer p, without a conflicting store in the middle, then the later statement will return the memoized value, x_r.

The last case deals with merging. This case applies at the usual control-flow merge points due to branching statements, as well as at implicit control-flow merge points added to deal with function calls. We define the intersection of two mappings $R_1 \cap R_1$ to be all pairs (a, b) such that $a \in R_1$, $a \in R_2$, and $R_1(a) = R_2(a) = b$.

Reasoning About Cross-Thread Interference. As described briefly in Section 2.5.4, we reason about cross-thread interference by labelling each memory locations in R_{heap} as either *conflict-free* or *shared*. A location is conflict-free if it is provably thread-local (as determined by a static escape analysis) or if all writes to the location must-occur before the first call to `threadCreate`. The second case captures a common idiom where the main thread initializes global data that is kept read-only during parallel execution.

Shared locations may have conflicts. We reason about these conflicts using *interference-free regions (IFRs)* [41]. The details are beyond the scope of this dissertation. Briefly, the concept of IFRs derives from the following observation: if a program is assumed to be data race free,[4] then after thread T reads memory location p, the value at location p cannot be mutated by another thread until T executes certain combinations of synchronization operations, such as a `lock()` followed by an `unlock()`. Hence, during this region of execution (from the read of p until the `unlock()`), p is *interference-free.*

We compute IFRs using the method described by Effinger-Dean [42], then use IFRs to remove reaching definitions from R_{heap} when they may become invalid due to cross-thread inteference. Specifically, given that an access to pointer p has an IFR extending from statement γ_1 to γ_2, we remove p from R_{heap}^{out} on γ_2's outgoing control-flow edges, as the value at location p may be mutated by some other thread immediately after statement γ_2 has executed. (This detail is not illustrated in Figure 2.17.)

Function Summaries. We make our dataflow analysis scalable by using function summaries in an entirely standard way: For each function f, we compute summaries R_{local}^{f} and R_{heap}^{f}. Expressions in these summaries use symbolic constants $\bar{x}$ that represent the unknown values of f's formal parameters, $\bar{r}$. To apply a summary at a call site, we substitute $\bar{x}$ with the call site's arguments $\bar{e}$. We compute function summaries with a single bottom-up traversal of the static call graph, and resolve strongly-connected components (recursive calls) using iteration until convergence. To illustrate the use of function summaries, consider the following example:

```
void main() {       void foo(r2) {      void bar(r2) {
   foo(41)             ...                 lock()
   foo(42)          }                      ...
   bar(5)                               }
}
```

Suppose our program context of interest begins with thread T about to execute the `lock()`

[4] Recall from Section 2.5 that we treat data races as errors in *SimpThreads*, so we can effectively assume race freedom, making an IFR analysis a valid approach.

call in `bar`. There is one path from program entry up to this initial context: `main:foo(41)`, `main:foo(42)`, `main:bar(5)`, `bar:lock()`. When analyzing this path, we use function summaries to evaluate the calls to `foo`. However, we cannot use a function summary to evaluate the call to `bar`, as `bar` does not return on this path. Instead, we use the rules from Figure 2.17: to evaluate the call of `bar`, we assign `r2=5` in R_{local} and advance to the first statement of `bar`.

Heuristics and Alternative Designs. We have found it profitable to perform loop unrolling within our dataflow analysis: we unroll a loop if its trip count is fixed to a small, constant value. Additionally, although our current implementation uses context-insensitive function summaries and no path sensitivity other than loop unrolling, more precise approaches are well-studied and can be substituted [36, 89].

2.7 Soundness and Completeness

Our symbolic execution algorithm is sound, and it is complete except when the SMT solver uses concretization to make progress through an unsolvable query (recall Section 2.3.1). We make this claim only for programs with a correctly implemented pthreads library; otherwise, the invariants from Section 2.5.3 would be incorrect. Further, as the initial symbolic state S_S is a given in the statement of our theorem (below), we implicitly assume correctness of the dataflow analyses used to construct that initial symbolic state.

Our theorem relies on a notion of correspondence between concrete and symbolic states—because the heap is expanded lazily in our symbolic semantics, this notion relies on *partial equivalence* and is somewhat technical. We give a full definition of partial equivalence and a proof of the theorem in Appendix B.

Definition 1 (Correspondence of concrete and symbolic states)**.** *We say that symbolic state S_S* models *concrete state S_K* under *constraint C if there exists an assignment Σ that assigns all symbolic constants in S_S to values such that (a) Σ is a valid assignment under the constraint C, and (b) the application of Σ to S_S produces a state that is* partially-equivalent *to S_K (as defined in Appendix B).*

Theorem 1 (Soundness and completeness of symbolic execution)**.** *Consider an initial program context, an initial concrete state S_K for that context, and an initial symbolic state S_S:*

– ***Soundness:*** *If symbolic execution from S_S outputs a pair (p, C), then for all S_K such that S_S models S_K under C, concrete execution from S_K must follow path p as long as context switches happen exactly as specified by path p.*

– ***Completeness:*** *If concrete execution from S_K follows path p, then for all S_S such that S_S models S_K under $S_S.C$, symbolic execution from S_S will either (a) output a pair (p, C), for some C, or (b) encounter a query that the SMT solver cannot solve.*

2.8 Implementation

We implemented the above algorithms on top of the Cloud9 [24] symbolic execution engine, which is an extension of Klee [27] that adds support for multithreaded processes. Cloud9 symbolically executes C programs that use pthreads and are compiled to LLVM [68] bitcode (Cloud9 operates directly on LLVM bitcode). Our modifications added about 4500 lines of C++ code. Where a points-to analysis is needed, we use DSA [67].

The C language allows casts between pointers and integers. This is not modeled in our semantics but is partially supported by our implementation. Our approach is to represent each pointer expression p like any other integer expression. Then, at each memory access, we analyze p to extract (*base*, *offset*) components. For example, our implementation represents `int *p = &a[x*3]` as $p = a + 4 \cdot (x \cdot 3)$, and to access p we transform it to $\texttt{ptr}(a, 12 \cdot x)$. We determine that a is the *base* address by exploiting LLVM's simple type system to learn which terms are used as pointers.

The precise semantics of integer-to-pointer conversions in C are implementation-defined (§6.3.2.3 of [55]). Our implementation does not support programs that use integer arithmetic to jump between two separately-allocated objects, such as via the classic "XOR" trick for doubly-linked lists. Such programs are not amenable to garbage collection for analogous reasons [18], even though they are supported by some C implementations.

2.9 Evaluation

2.9.1 Infeasible Paths

Recall from Section 2.1 that our approach lies on a spectrum between a *naïve* approach, which approximates the initial state very conservatively by leaving all memory locations unconstrained, and a *fully precise* approach, which constructs a perfectly precise initial state using an intractably expensive analysis. In this section, we attempt to characterize how close our approach lies to both ends of this spectrum.

We first compare the *naïve* approach with our approach: *how many fewer infeasible paths do we explore?* We answer this question for a given program context C by exhaustively enumerating all paths reachable from C up to a bounded depth. Any path that is enumerated by the *naïve* approach, but not by our approach, *must* be an infeasible path that our approach has avoided. We use a bounded depth to make exhaustive exploration feasible.

Table 2.1 summarizes our results. Each row summarizes experiments for a unique program context. We evaluated our implementation using five applications: `blackscholes` [16], which uses fork-join parallelism; `dedup` [16], which uses pipeline parallelism; `lu` [95] and `streamcluster` [16], which use barrier-synchronized loops; and `pfscan` [44], which uses task parallelism. These applications were selected because they cover a range of parallelism styles. For each application, we manually selected one or two program contexts in which at least two threads begin execution from the middle of a core loop. Column 2 shows the number of threads used in each initial context, and Column 3 shows the maximum number of conditional branches executed on each path during bounded-depth exploration.

Columns 4 and 7 show the number of paths explored by our fully optimized approach (*Full*) and the *naïve* approach, respectively. To further characterize our approach, we also ran our approach with optimizations disabled: *-RD* disables reaching definitions (recall Section 2.3.3, Section 2.4.2, Section 2.5.4, and Section 2.6.2) and *-SI* disables synchronization invariants (recall Section 2.5.3 and Section 2.5.4). Our approach explores significantly fewer infeasible paths compared to the *naïve* approach, and a comparison across Columns 4–7 shows that each optimization is essential.

Program Context			Num Paths				inf.
	thr	br	*Full*	*-RD*	*-SI*	*N*	pths
blackscholes	4	20	763	1087	765	1087	–
dedup-1	5	10	103	122	863	971	–
dedup-2	5	12	458	550	1811	1904	–
lu-1	4	22	681	1026	1133	1864	625
lu-2	4	18	554	1400	1290	4680	380
pfscan	3	18	246	246	3785	3785	–
streamcluster	3	11	60	617	229	1004	48

Table 2.1: Evaluation of infeasible paths enumerated during symbolic execution. *Full* is our fully optimized approach, and *N* is the *naïve* approach. *-RD* and *-SI* remove the *reaching definitions* and *synchronization invariants* optimizations.

It is difficult to compare our approach with the *fully precise* approach, as the *fully precise* approach is intractable. For `lu` and `streamcluster`, we have manually inspected the paths explored by our approach (Column 4) and estimated, through our best understanding of the code, how many of those paths are infeasible (Column 8). Sources of infeasible paths include the following: Both programs assign each thread a unique *id* parameter (*e.g.*, by incrementing a global counter), but we are unable prove that these *id*s are unique across threads, leading to infeasible paths. In `lu`, the *id* is computed by incrementing a global counter while a lock is held, and in `streamcluster`, the *id* is passed to the thread as an argument to `pthread_create`. We suspect that a similar situation causes infeasible paths in other applications, but we have not quantified this precisely. Further, they performs calls of the form `pthread_join(t[i])`—we are unable to prove that each `t[i]` is a valid thread id, so we must fork for (infeasible) error cases.

2.9.2 Evaluation: Performance

Now we answer a second question: *how does our approach affect the performance of symbolic execution?* We answered this question by measuring performance of the exhaustive explorations done above.

Table 2.2 presents these results. Columns 4–8 show the average number of LLVM instructions executed per second (IPS), and Columns 9–13 show the percentage of total execution

Program Context			Avg IPS					Exec Time in *isSat*				
	thr	br	*WP*	*Full*	*-RD*	*-SI*	*N*	*WP*	*Full*	*-RD*	*-SI*	*N*
blackscholes	4	20	927	176	1171	178	1206	75%	93%	65%	93%	65%
dedup-1	5	10	4731	72	49	67	64	3%	30%	62%	36%	51%
dedup-2	5	12	4692	45	26	39	32	5%	35%	64%	30%	59%
lu-1	4	22	3997	93	170	64	107	2%	55%	16%	75%	57%
lu-2	4	18	3860	80	136	105	162	32%	57%	23%	56%	26%
pfscan	3	18	6250	5368	5650	5254	5503	17%	28%	25%	15%	13%
streamcluster	3	11	5382	161	59	7	19	15%	9%	35%	74%	31%

Table 2.2: Evaluation of symbolic execution performance. This uses the same contexts and configurations as in Table 2.1. *WP* estimates the performance of an intractable but fully precise approach.

time devoted to *isSat*. The two metrics are correlated, as slower *isSat* times lead to lower IPS. *Full* uses more precise constraints than the *naïve* approach, but this does not necessarily lead to higher IPS for *Full*. Namely, precise and simple constraints such as $x = 5$ lead to high IPS, but precise and complex constraints can lead to low IPS—the latter effect has been observed previously [52, 61].

To further understand the overheads of our approach, we symbolically executed multiple *whole program* paths that each begin at program entry and pass through the initial context (*WP* in Columns 4 and 9). Although *Full* can be an order-of-magnitude slower than *WP*, many paths explored by *WP* visit 100s of branches before reaching the initial context, suggesting that exhaustive summarization of all paths from program entry is infeasible—approximating the initial context is *necessary.* Further profiling shows that much of our overhead comes from resolving symbolic pointers: LLVM's `load` and `store` instructions typically comprised 15% to 50% of total execution time in *Full*, but $< 5\%$ in *WP*.

Lastly, we tried disabling our use of a points-to analysis to restrict aliasing (Section 2.4.1, Section 2.5.2). With this optimization disabled, each symbolic pointer was assigned 100s of aliases, leading to large heap-update expressions and poor solver performance—so slow that on most benchmarks, throughput decreased to well under 5 IPS. Hence, we consider this optimization so vital that we left it enabled in all experiments.

Chapter 3

INPUT-COVERING SCHEDULES

The previous chapter described a way to limit path explosion in symbolic execution by focusing on small fragments of execution. This chapter attacks the problem in a more fundamental way by introducing the notion of *input-covering schedules*: given a program P, we say that a set of schedules Σ *covers* the program's inputs if, for all inputs, there exists some schedule $S \in \Sigma$ such that P's execution can be constrained to S and still produce a semantically valid result.

This chapter describes, first, an algorithm to enumerate input-covering schedules, and second, a way to exploit input-covering schedules using a custon runtime system. We start by overviewing our goals and solutions (Section 3.1). We then dive into the technical details. We start by describing our representation of *schedules*—this representation was carefully designed, as a poor representation makes the problem intractable (Section 3.2). Next, we describe our algorithm for enumerating input-covering schedules (Section 3.3), and to further connect this chapter with the prior chapter, we show how our symbolic execution techniques from the prior chapter are directly applicable in this new context (Section 3.3.3). We then describe a number of optimizations to our algorithm (Section 3.4 and Section 3.5).

We have implemented our system on the Cloud9 [24] symbolic execution engine, and further, we have implemented a custom runtime system that constrains execution to input-covering schedules produced by our algorithm. We have focused our work on the fundamental problem of enumerating input-covering schedules—our schedule-enumeration algorithm includes many optimizations, but our custom runtime system is a proof-of-concept that has not been fully optimized. We describe these implementations (Section 3.6) and then summarize our system as a whole by giving a formal statement of the guarantees provided by our system (Section 3.7).

We have performed the first empirical evaluation to address the fundamental question: "How large are sets of input-covering schedules?" We end by discussing this evaluation (Section 3.8). We organize our evaluation as a set of case studies to carefully characterize the program analysis challenges inherent to enumerating input-covering schedules for realistic multithreaded C programs. We show that we are able to enumerate input-covering schedules for some programs, and for other programs, we characterize the ways in which our analysis is imperfect.

3.1 System Overview

Our system works as follows: Given a program P, we first enumerate a small input-covering set Σ using symbolic execution. As stated above, a set of schedules is *input-covering* if it contains enough schedules to enable correct program execution. That is, for each possible input i, there must exist some schedule $S \in \Sigma$ such that, when program P is given input i, P's execution can be constrained to S and still produce a semantically valid result.

After enumerating Σ, we attach a custom runtime system to P that *constrains execution* to follow only those schedules in Σ. This combination of program and runtime system is essentially a new program, P', that accepts all possible inputs and produces semantically correct behavior, like the original program, but uses fewer schedules. We always run the constrained program P' in deployment. The result is that Σ contains the *complete* set of schedules that might be encountered during deployment—this simplifies the verification problem by reducing the number of schedules that must be considered.

It is not obvious that small sets of input-covering schedules should exist for realistic multithreaded programs. The key word is *small*—an input-covering set Σ is of no help when it is so intractably large that it cannot be enumerated in a reasonable time. Moreover, we ideally want to find the *smallest possible* set of input-covering schedules to minimize the amount of work that must be done during program testing and verification. An important contribution of this work is defining Σ in a way that makes finding small input-covering sets more tractable. Notably, programs that run for unbounded periods of time can require unboundedly many schedules, making the set Σ intractably large. We avoid this problem by partitioning execution into *bounded epochs*—we find input-covering schedules for each

```
input X
global Lock A,B

Thread 1                 Thread 2
for (i in 1..5) {        for (i in 1..5) {
  if (X == 0) {            if (X == 0) {
    lock(A)                  lock(A)
    unlock(A)                unlock(A)
  } else {                 } else {
    lock(B)                  lock(B)
    unlock(B)                unlock(B)
  }                        }
}                        }
```

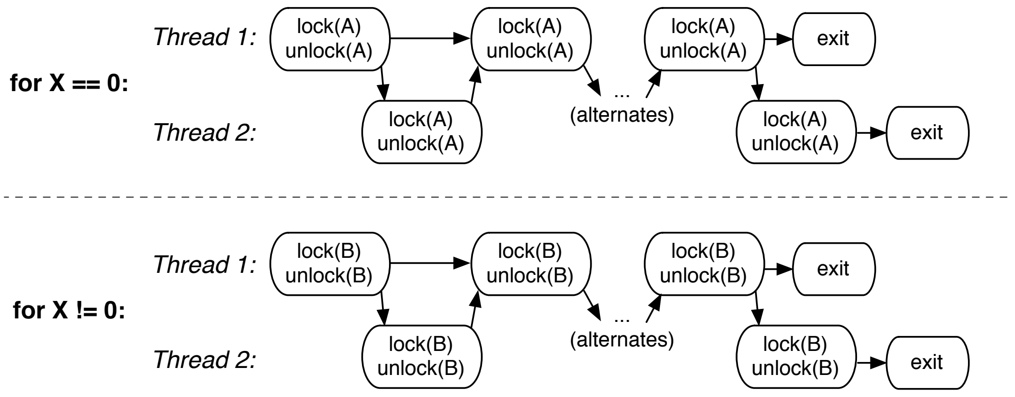


Figure 3.1: On top is a simple multithreaded program. On the bottom is one set of input-covering schedules for the program.

epoch in isolation, and then piece those schedules together at runtime.

Hence, our system contains three components: a schedule enumerator, a runtime system, and a program verification strategy. Below, we summarize each component in slightly more detail. We give complete descriptions in later sections of this chapter.

Enumerating Σ. We use an algorithm based on symbolic execution to systematically enumerate input-covering schedules. Figure 3.1 gives a demonstration. On the bottom of Figure 3.1 is a set of input-covering schedules, Σ, that our algorithm might produce when given the program on the top. Each schedule in Σ is paired with an *input constraint* that describes the set of inputs under which the schedule can be followed. Schedules are specified as happens-before orderings of dynamic instances of synchronization statements.

Runtime System. At runtime, we constrain execution to follow schedules in Σ. We have implemented a custom runtime system that captures the program's input, finds a pair (I,S) $\in \Sigma$ such that the program's input satisfies input constraint I, and then constrains execution to S, ensuring that execution never deviates from S.

Verification Strategy. Finally, testing and verification become simpler under the assumption that programs always execute using our custom runtime system. Given this assumption, the input-covering set Σ contains the *complete* set of schedules that might be followed at runtime, and as a result, verification tools can focus on schedules in Σ only, avoiding the need to reason about a massive nondeterministic interleaving space.

For a simple example, consider deadlocks. We can determine if a schedule deadlocks by simply looking at it—if any thread does not terminate with an exit statement, then the schedule deadlocks. We can perform this check for each schedule in Σ independently. If a deadlocking schedule is found, we can use the schedule's associated input constraint to present the programmer with a concrete input and schedule that leads to deadlock. If no deadlocking schedules are found, we have *proven* that we will never encounter a deadlock when execution is constrained by our runtime system. We have built a simple deadlock checker that we describe in Section 3.6.3.

More generally, we can reason about each schedule in isolation by serializing the original multithreaded program into $|\Sigma|$ single-threaded programs, where each single-threaded program P_i is constructed by serializing the original multithreaded program P according to schedule $S_i \in \Sigma$. This reduces the worst-case number of possible program behaviors from $O(k! \cdot i)$ to $O(|\Sigma| \cdot i)$, where k is the length of execution and i is the number of possible inputs. This approach is called *schedule specialization* and it has been shown to have real benefits [96]. For example, consider the following code:

```
Thread 1         Thread 2
lock(L)          lock(L)
if (x%2!=0)      if (x%2!=0)
  x++              fail()
unlock(L)        unlock(L)
```

There is an assertion failure in T_2 when it executes before T_1 with an odd value for `x`. This bug can be difficult to find in conventional systems because it depends on specific combinations of input (`x`) and schedule (ordering of lock acquires). Our approach computes just one schedule for this code snippet (say, T_1 before T_2), which reduces the verification problem from a hard thread interleaving problem to a simpler (but still difficult) single-threaded reachability problem. We refer to Wu *et al.* [96] and Yang *et al.* [98] for more detailed arguments in favor of schedule specialization.

Assumptions. Our schedule enumeration algorithm assumes data race freedom. When this assumption is broken, we do not compute a true input-covering set—execution may diverge from the expected schedule after a data race. We make this assumption to simplify our analysis in a number of important ways that will be mentioned later.

We also assume that each program has a bounded number of live threads at any given moment. If the number of live threads is input-dependent, we expect the programmer to supply an upper bound for that input. Bounding the number of threads allows us to represent each thread explicitly in the schedule, as shown in Figure 3.1.

3.2 Representing Schedules

We represent each schedule as a happens-before graph over a finite execution trace, where graph nodes are labeled by the triple *(program-counter, thread-id, dynamic-counter)* and edges are induced from program order and synchronization in the usual way, such as between release and acquire operations on the same lock. The *program-counter* label represents a synchronization statement in the program, such as a call to `pthread_mutex_lock`, and the pair *(thread-id, dynamic-counter)* is a Lamport timestamp [65]. Notice that ordinary memory accesses are not included in the happens-before graph, as we assume data race freedom.

We return to the example in Figure 3.1. On top is a simple program in which each thread acquires a different global lock depending on the value of the input `X`. A conventional nondeterministic execution might follow one of 240 possible schedules (5! when `X==0`, and another 5! when `X!=0`). However, just *two* schedules are necessary to cover all inputs for this program—one schedule for `X==0`, and another for `X!=0`. This is illustrated by the bottom half of Figure 3.1, which shows one possible set of input-covering schedules, Σ. (The schedules have been abbreviated for space.)

Importantly, for each pair (I,S) $\in \Sigma$, the constraint I should include only those conditions that affect whether the schedule S can be followed. That is, constraint I should be a *weakest precondition* of the schedule S. For example, suppose we modify the program in Figure 3.1 to perform a complex computation in each loop iteration. As long as this computation does not mutate `X` or perform synchronization, the set of input-covering schedules shown in Figure 3.1 will be equally correct for our modified program.

The above representation works well for programs that read their entire input up front (*e.g.*, from the command-line or a file) and then perform a bounded-length computation on that input. We now extend our representation of Σ to support more realistic programs that read dynamic inputs (Section 3.2.1) and execute for an unbounded time (Section 3.2.2).

3.2.1 Programs That Read Inputs Dynamically

For programs that read inputs dynamically, it is not possible to select a complete schedule at the beginning of execution because some inputs are not yet available. Following Tern [34], we support such programs by representing Σ using a schedule tree, where each edge of the tree represents a *partial* happens-before schedule.

We use the schedule tree during runtime execution as follows. At the beginning of the program, we select a pair (I,S) from the root of the tree, where I is an input constraint that matches the input available at program entry, and where S is a partial schedule. We follow S until the program reads more input, at which point we advance to a child node in the schedule tree, and select a pair (I',S') from that node such that I' is a constraint that matches the input that is now available. We continue along the partial schedule S', and repeat this process until reaching a leaf of the schedule tree, at which point the program will either deadlock or exit.

For simplicity and brevity, most of this chapter uses the simplifying assumption that programs do not read inputs dynamically, since such inputs do not introduce any challenges beyond those mentioned here. Thus, from here on, the symbol Σ will refer to a simple set of pairs (I,S), though in practice it is a tree as described above. We will return to dynamic inputs when describing our runtime system in Section 3.6.2.

3.2.2 Bounded Epochs

We support programs of unbounded length by partitioning execution into *bounded epochs.* In practice, we care not only about programs of truly unbounded length, but also about programs that execute for a "very long" time. For example, consider the following simple program with two threads:

```
Thread 1             Thread 2
for (i in 1..X) {    for (i in 1..Y) {
  lock(L)              lock(L)
  unlock(L)            unlock(L)
}                    }
```

If `X` and `Y` are program inputs, then any set of input-covering schedules must have a unique schedule for each pair (`X`,`Y`). If `X` and `Y` are 32-bit integers, there are 2^{64} possible inputs, so any set of input-covering schedules *must* contain 2^{64} total schedules. Equally problematic: the longest of these schedules must contain 2^{64} total synchronization operations.

Our basic idea is to define schedules one loop iteration at a time. We do this by partitioning the program into bounded epochs that are separated by *epoch markers.* We statically analyze the program to find all loops that may perform synchronization, and then place an epoch marker at the entry of such loops. The details of this process are explained in Section 3.3.1. In short, the above program would be annotated as follows:

```
Thread 1             Thread 2
for (i in 1..X) {    for (i in 1..Y) {
  epochMarker()        epochMarker()
  lock(L)              lock(L)
  unlock(L)            unlock(L)
}                    }
```

Epoch markers act as barriers during program execution, forcing threads to execute in a bulk-synchronous manner. For example, suppose a program's threads begin executing from some initial state. The threads will execute concurrently until each thread is blocked on synchronization, has terminated, or has reached a future epoch marker (possibly the same epoch marker the thread started at, *e.g.*, if the thread went back around the same loop). This quantum of execution corresponds to a single bounded epoch. Execution repeats in this bulk-synchronous manner until all threads terminate. We include "is blocked" in the end-of-epoch condition to avoid deadlock when thread T_1 attempts to acquire a lock that is held by T_2 while T_2 is stalled at an epoch marker. Note that, in practice, we can use loop unrolling to reduce the frequency of epoch markers (see Section 3.5).

We now require a set of input-covering schedules for each bounded epoch. A bounded epoch E is named by a list of pairs $(pc_i, callstack_i)$, where pc_i represents the current program counter of thread T_i (*i.e.*, the pc of an epoch marker) and $callstack_i$ is a list of return addresses that represents the calling context. Our algorithm, defined in Section 3.3, enumerates all reachable bounded epochs $\mathcal{E}$ and computes an input-covering set Σ_E for each E $\in \mathcal{E}$. The initial bounded epoch starts at program entry, and its inputs are the program's inputs. All other epochs start from a point in the *middle* of a program's execution. The "input" to these epochs is, potentially, the entire state of memory, which introduces challenges for our runtime system that we will address in Section 3.6.2. (This definition of epochs also hints at why our techniques from Chapter 2 will become essential.)

3.2.3 Discussion

Bounded epochs make an intractable problem tractable—they limit combinatorial explosion by bounding both the length of each computed schedule as well as the total number of schedules—but they introduce necessary approximations, as we will demonstrate in Section 3.3.3. Further, bounded epochs do not eliminate all causes of explosion in the size of Σ. For example, consider a thread that determines which locks to acquire using a sequence of conditionals as in the following:

```
Thread 1
if (X[0]) {     if (X[1]) {          if (X[n]) {
  lock(L[0])      lock(L[1])    ...    lock(L[n])
  unlock(L[0])    unlock(L[1])         unlock(L[n])
}               }                    }
```

In this case, the set of locks acquired by thread T_1 is uniquely determined by the value of the bitvector `X`. If `X` has 32 bits, any set of input-covering schedules *must* have 2^{32} unique schedules. This is a source of explosion in the size of Σ that we can think of no good way to eliminate. Since our underlying problem is undecidable, anyway, we focus our current work on programs without such pathological behavior.

Challenges. Bounded epochs introduce two challenges that we will address in Section 3.5. First, how can epochs be made performant? Since epochs are runtime barriers, a concern is *imbalance* of work across threads.

Second, since schedules terminate at epoch boundaries, how can verification be effective? We observe the following: any bug that can be detected by examining a single point of execution can be detected by examining a single epoch in isolation, perhaps by using schedule specialization on each epoch. Bugs identifiable from a single point of execution include deadlocks and assertion failures. However, as we will describe shortly, some schedules may actually be *infeasible*—leading to false-positives—and some mechanism of detecting those infeasible schedules is desirable. Other bugs can be detected only by examining sequences of instructions. Atomicity violations are one such example. To simplify detection of these bugs, epochs should be long enough so that *most* buggy instruction sequences will be contained within either one epoch or one short sequence of epochs.

3.3 Finding Input-Covering Schedules

Our algorithm for enumerating input-covering schedules is shown in Figures 3.2–3.4. The input is a program P, and the output is a mapping from epochs $\mathrm{E} \in \mathcal{E}$ to a set of input-coverings schedules Σ_E for each epoch, where $\mathcal{E}$ is a set of bounded epochs that *may* be reachable.

We first invoke `PlaceEpochMarkers` to instrument the program with epoch markers. We then invoke `Search` to traverse all reachable bounded epochs, starting from an initial epoch representing the call to `main()`. For each epoch E, `Search` invokes `SearchInEpoch(E)`, which performs a depth-first search to enumerate a set of input-covering schedules for E along with the set of epochs reachable from E. We describe each function below.

3.3.1 Placing Epoch Markers

The basic constraint for epoch marker placement is the following: we must ensure that a bounded number of synchronization operations are executed before the program either reaches another epoch marker or halts. This ensures that schedules cannot grow to an unbounded length.

```
1  PlaceEpochMarkers(p: Program) {
2    covered = {"epochMarker","pthread_barrier_wait"}
3    workqueue = {all functions that directly perform synchronization}
4    while (!workqueue.empty()) {
5      F = workqueue.popfront()
6      foreach (loop L in F, bottom-up) {
7        if (L may perform synchronization
8            && !IsTrivialLoop(L)
9            && $\nexists$ epoch marker that must-execute in L)
10         place epoch marker at L.entry
11     }
12     if ($\exists$ epoch marker that must-execute in F)
13       covered.add(F)
14     workqueue.pushback(immediate callers of F)
15   }
16 }
```

Figure 3.2: Algorithm to place epoch markers

Naïvely, we could satisfy this requirement by placing epoch markers in all loops that *may* perform synchronization, including loops that perform synchronization either directly (*e.g.*, by calling `pthread_mutex_lock`) or indirectly (*e.g.*, by calling a function that transitively calls `pthread_mutex_lock`).[1] However, it is important to minimize the number of epoch markers—a large number of epoch markers can lead to a large number of epochs in $\mathcal{E}$.

Our actual algorithm, `PlaceEpochMarkers`, is more careful. We use a bottom-up traversal of the call graph starting from functions that directly perform synchronization (lines 3–5 and 14). For each visited function, we place epoch markers in all loops that perform synchronization and are not pruned by one of the following three optimizations:

Ignore trivial loops. We ignore simple loops of the form:

```
while (!condition)
  pthread_cond_wait(cvar, mutex)
```

This is the common idiom for using pthreads condition variables. We observe that this loop can execute an unbounded number of synchronization operations only if the condition

[1] Our current implementation does not support recursive functions that synchronize. This can be remedied by transforming recursive functions into equivalent iterative functions, either manually, or automatically as in [71].

variable `cvar` can be notified an unbounded number of times. So, as long as we ensure that all loops containing notifications are covered by an epoch marker, we can avoid placing an epoch marker in the above loop.

Similarly, we ignore loops where the only form of synchronization is a call to `pthread_create` or `pthread_join`. These loops must be bounded since we assume a bounded number of threads are live at any given moment (recall Section 3.1).

Don't Cover the Same Loop Twice. We consider a loop *covered* when there exists an epoch marker that must-execute on each iteration of the loop. For example, if loop A contains loop B where B contains an epoch marker, and if at least one iteration of B must-execute for each iteration of A, then we can avoid placing an epoch marker in loop A because that is subsumed by the marker placed in loop B.

We implement this optimization by visiting the loop forest bottom-up (line 6). Then, we ignore each loop that must-execute a previously placed epoch marker (line 9). The variable `covered` contains a set of functions that *must* execute an epoch marker, so the check at line 9 is implemented by checking if there must-exist a call to a function in the `covered` set—notice that at line 2, we initialize `covered` to include the `epochMarker` function.

Barriers are epoch markers. Since epoch boundaries are runtime barriers, we might as well end epochs at explicit program barriers. So, at line 2, we initialize `covered` to include `pthread_barrier_wait` so the optimization at line 9 will ignore loops that must-execute a barrier. In this way, each call to `pthread_barrier_wait` is treated as an implicit epoch marker.

3.3.2 Enumerating Schedules for a Single Epoch

The function `SearchInEpoch` (Figure 3.4) uses `ExecutePath` to symbolically execute a single path from a given initial state. This path completes when all threads have reached an epoch marker, terminated, or deadlocked. `ExecutePath` produces a final symbolic state along with an execution trace. `ExecutePath` can follow any path and may context switch between threads arbitrarily, as long as it follows a path that is feasible given the initial input constraint. If the path did not end in program termination or deadlock, it ended at

```
let hd = thread.slice.head in
if (!Postdominates(hd, branch)
      || WritesLiveVarBetween(branch, hd)
      || SyncOpBetween(branch, hd))
  Take(branch)
```

Figure 3.3: How precondition slicing handles branches (our additions are in *italics*)

a new bounded epoch that we add to the set of reachable epochs (lines 18–19). `EpochId` extracts the epoch identifier (recall from Section 3.2.2 that an epoch is named by the calling contexts from which each of its threads begins execution).

For each path, we extract the schedule and then compute a conservative weakest precondition of the schedule using precondition slicing [32], where a *precondition slice* is computed from an execution trace and includes only those statements from the trace that might affect whether the final statement was executed. The set of branching statements in a precondition slice combine to form a *precondition* of the final statement. We have modified the algorithm from Costa *et al.* [32] to instead enumerate all statements from the trace that might affect the set of synchronization operations that would be performed. We call this a *synchronization-preserving slice.*

The original algorithm in [32] works much like a standard dynamic backwards slicing algorithm: it iterates backwards over an execution trace, uses a *live* set to track data dependencies, and adds statements to the slice if they modify items in the *live* set. Branches are handled as shown in Figure 3.3: a branch is included in the slice if either (a) the current head-of-slice is control-dependent on the branch (this is the `Postdominates` check, which is computed with a standard postdominators analysis), or (b) some other path through the branch (not taken in the given trace) might modify an item in the *live* set (this is the `WritesLiveVarBetween` check, which is computed with a static alias analysis). We refer to the paper by Costa *et al.* for a more complete description of the original algorithm [32].

We make three modifications. First, we include all synchronization statements in the slice to ensure that all control and data dependencies of synchronization are included in the slice. Second, we construct a separate slice for each thread so that all control-flow checks

```
SearchInEpoch(initState: SymbolicState) {
  reachableEpochs = {}
  schedules = {}
  constraints = {true}

  while (!constraints.empty()) {
    // Explore a new input constraint
    state = initState.clone()
    state.applyConstraint(constraints.remove())
    (finalState, trace) = ExecutePath(state)

    // Update set of schedules
    slice = PrecondSlice(trace)
    schedules.add(MakeConstraint(slice.branches),
                  trace.schedule)

    // Update set of reachable epochs
    if (!IsTerminatedOrDeadlocked(finalState))
      reachableEpochs.add(EpochId(finalState))

    // Accumulate unexplored input constraints
    inputConstraint = true
    for (b in slice.branches) {
      c = inputConstraint ∧ ¬b
      if (c not yet covered)
        constraints.add(c)
      inputConstraint = inputConstraint ∧ b
    }
  }

  return (schedules, reachableEpochs)
}
```

Figure 3.4: Enumerating schedules within a single epoch

in Figure 3.3 remain single-threaded. Finally, we include a branch in the slice if some other path through the branch (not taken in the given trace) might perform synchronization (this is the `SyncOpBetween` check in Figure 3.3). The final addition ensures that a branch is included in the slice if it may affect synchronization.

Because we assume data race freedom, our slicing algorithm does not need to account for potentially-racing accesses when computing data dependencies. Relaxing this assumption would involve a much more complicated implementation of `WritesLiveVarBetween` that would require a conservative may-race analysis, as we describe in Section 4.2.

Shortest-Path First. It is correct for `ExecutePath` to follow any feasible path. However, it is optimal for `ExecutePath` to execute the *shortest* feasible path—longer paths should be executed only as necessary to cover inputs not covered by the shortest path. Determining the true shortest feasible path is not decidable, so at each branch our heuristic is to select the branch edge with the shortest static distance to a statement that either returns from the current function or exits the current loop.

3.3.3 Exploring All Reachable Epochs

The function `Search` enumerates input-covering schedules for all epochs that are uncovered by `SearchInEpoch`. In `Search`, the key is a call to `MakeStateForEpoch`, which computes, for a given epoch, an initial symbolic state that will be explored by `SearchInEpoch`. Each symbolic state includes a set of calling contexts (one calling context per thread), along with a set of constraints on memory. The calling contexts are provided directly by the epoch identifier, but the memory constraints must be computed by `MakeStateForEpoch`.

How does `MakeStateForEpoch` compute the initial memory constraints? The answer is given in Chapter 2 of this dissertation: we use a context-specific dataflow analysis to construct an initial symbolic state that over-approximates all possible concrete initial states. Then, `ExecutePath` (see line 10 of Figure 3.4) performs symbolic execution using the symbolic execution semantics described in Chapter 2. As each epoch identifier contains complete calling contexts for each thread, we do not need to deal with an underspecified calling contexts (recall Section 2.3.2).

```
Search(p: Program) {
  worklist = {MakeInitialState(p)}
  output = {}

  while (!worklist.empty()) {
    // Explore another bounded epoch
    state = worklist.remove()
    (schedules, reachable) = SearchInEpoch(state)

    // Found an input-covering set for this epoch
    output.add(EpochId(state), schedules)

    // Add unexplored epochs to the worklist
    for (e in reachable)
      if (e not yet visited)
        worklist.add(MakeStateForEpoch(e))
  }

  return output
}
```

Figure 3.5: Exploring all reachable epochs

3.4 Avoiding Combinatorial Explosion

Avoiding combinatorial explosion is essential. This section describes two categories of optimizations:

First, we define optimizations that exploit *redundant schedules* (Section 3.4.1). These optimizations allow us to cover more inputs with fewer schedules. Precondition slicing can be viewed as one such optimization, but the optimizations in Section 3.4.1 go further by observing that schedules that are not obviously the same can sometimes be treated as if they are.

Second, we deal with unbounded loops that contain synchronization using bounded epochs, but what about unbounded loops that do not contain synchronization? We are hesitant to place epoch markers in every loop since a large number of epoch markers can lead to a large number of epochs. Instead, we deal with unbounded synchronization-free loops using a technique we call *input abstraction* (Section 3.4.2).

3.4.1 Pruning Redundant Schedules

3.4.1.1 Ignoring Prefix Schedules

Programs are often implemented using a defensive coding style: they frequently check for errors (*e.g.*, via assertions or by checking return codes from system calls) and terminate the program when a failure is detected. Since we include "thread exit" events in our schedules, it appears that enumerating a complete set of input-covering schedules requires enumerating all ways in which the program can exit. In the limit, this requires enumerating all feasible assertion failures, which is a very hard problem on its own.

We avoid this problem using the concept of *prefix schedules*. Suppose a thread executes the following code fragment:

```
lock(A)
if (X == 0) { abort() }
lock(B)
```

Concretely, there are two feasible schedules: (1) the thread locks `A` and then aborts the process, and (2) the thread locks `A` and then locks `B`. We consider the first schedule a *prefix* of the second schedule: at runtime, execution can always follow the second schedule, and then stop early if the `abort` statement is reached. To support prefix schedules, we modify `ExecutePath` and `PrecondSlice` to ignore branches that exit the process before performing any synchronization. For the above fragment, our optimized algorithm outputs just the second schedule, paired with the input-constraint *true*. We arrive at this output by ignoring the *true* branch of `if(X==0)`. Note, however, that we would require two schedules if there was a call to `lock()` just before the `abort()`.

3.4.1.2 Ignoring Library Synchronization

Users of our tool can opt to ignore internal synchronization used by library functions such as `printf` to ensure consistency of internal library data structures. With this option, our algorithm produces schedules that do not include internal library synchronization—such synchronization will be performed nondeterministically at runtime. Our rationale is that developers are more concerned about testing their own code than library internals, so it is sensible to ignore library internals and construct input-covering schedules for application code only. This option works especially well with the prefix schedules optimization (Section 3.4.1.1), as programs often call `printf` just before aborting the program.

3.4.1.3 Symbolic Thread Ids

Redundant schedules can also arise across epoch boundaries. For example, consider a program in which N threads each execute the following code:

```
1 while(X) { epochMarker() ... }
2 ...
3 epochMarker()
```

If `X` evaluates differently for each thread, then naïvely, we need one epoch in which all threads start at the epoch marker at line 1, another in which T_1 starts at line 3 while all other threads start at line 1, another in which just T_2 starts at line 3, another in which just T_1 and T_2 start at line 3, and so on. Hence, since there are N threads and each thread can start from one of two statements containing epoch markers (line 1 or line 3), then in total, there are 2^N epochs.

The above combinatorial explosion arises if we assign each thread a *concrete* thread id during symbolic execution. We can avoid this problem by instead assigning each thread a *symbolic* thread id during symbolic execution. Now, we need to consider just $N+1$ total epochs: one epoch in which all threads start at line 1, another epoch in which one thread starts line 3, another in which two threads start at line 3, and so on. The idea is that the specific assignment of thread ids to calling contexts does not matter, so the `EpochId` function should return a *multiset* of calling contexts, rather than an ordered list. This optimization requires some cooperation with our runtime system. Specifically, at each runtime epoch boundary, we must dynamically map each symbolic thread id to a concrete thread id—we defer details to Section 3.6.2.

More generally, if we extend the above example to use k epoch markers, then we explore k^N epochs using concrete thread ids, and just $\left(\!\binom{k}{N}\!\right)$ epochs using symbolic thread ids, where $\left(\!\binom{k}{N}\!\right)$ is k-choose-N with repetitions. However, if we further modify the above example so that each thread T_i executes a unique function f_i, where each f_i includes k epoch markers, then we must explore k^N epochs because there are that many unique combinations of calling contexts—this sort of combinatorial explosion is fundamental to our definition of epochs, so it cannot be avoided.

3.4.1.4 Redundancy from Code Duplication

We run our schedule enumeration algorithm *after* a compiler optimization pass, as this has been shown to speed-up symbolic execution [27]. However, optimizations can sometimes introduce schedule redundancies by duplicating synchronization. One example is the following transformation, which is called *jump threading* in LLVM:

```
if (X == 0) { f() }         if (X == 0)
lock()                 =>   { f(); lock(); g() }
if (X == 0) { g() }         else { lock() }
```

Our algorithm starts by executing one path through the optimized code (on the right). Suppose we execute the *false* branch. Precondition slicing will notice the `lock()` call in the *true* branch and direct us down that path as well, and the end result is an input-covering set with two schedules, one for `X==0` and `X!=0`. However, both of these schedules are the *same* schedule—the choice of schedule has no real dependency on input `X`.

Our current approach is to disable all transformations that might duplicate code, but unfortunately, this is not always possible. Notably, we cannot disable the following loop transformation because it is fundamental to the way many compilers reason about loops:

```
while (foo()) {        if (foo()) {
  ...            =>      do { ... } while (foo())
}                      }
```

If `foo()` performs synchronization or contains an epoch marker, duplication of the call to `foo()` can lead to redundant schedules that we cannot avoid.

3.4.1.5 Conservative Happens-Before Schedules

Consider the following example, where it is known that `B!=C` but it is not known whether `A==B` or `A==C`. For this example, assume `ExecutePath` executes threads T_2 and T_3 before T_1:

```
Thread 1        Thread 2        Thread 3
lock(A)         unlock(B)       unlock(C)
```

The precise happens-before schedule depends on whether lock `A` aliases locks `B` and `C`, and in fact, a complete set of input-covering schedules needs at least three happens-before schedules: one for `A==B`, one for `A==C`, and one for `A!=B && A!=C`.

We can cover the above example with just one schedule by constructing an *approximate* happens-before schedule that draws happens-before edges to `lock(A)` from both `unlock(B)` *and* `unlock(C)`. This results in fewer schedules in Σ at the cost of over-synchronization at runtime. Interestingly, the authors of Tern [34] observed that some programs perform well even when *all* synchronization is serialized, suggesting that this optimization's cost can be acceptable.

3.4.2 Abstracting Input Constraints

Unbounded synchronization-free loops can cause an explosion in the number of paths explored by `SearchInEpoch`. The following code fragment is a good example:

```
TreeNode* T = TreeBinarySearch(x)
if (T) { lock(L) ... }
```

The above code searches for a value in a binary tree using the standard recursive algorithm, and then performs synchronization if the value is found. Our problem is that symbolic execution will eagerly enumerate *all* concrete trees for which the expression `T!=NULL` evaluates to true. Specifically, it attempts to enumerate the following infinite set of input constraints:

```
root->x == x
root->x > x && root->left && root->left->x == x
root->x > x && root->left && root->left->x > x && root->left->left && ...
...
```

Our approach is a form of abstraction: instead of executing `TreeSearch` symbolically, we treat `TreeSearch` as an *uninterpreted function* and add `TreeSearch(x)!=NULL` to the path constraint. There are two subtleties in this approach:

What Do We Abstract? We abstract all synchronization-free loops and recursive functions that produce *live-out* values that might affect synchronization. Notice that we do *not* abstract loops that contain synchronization, since those loops are already bounded by epoch markers. We start by assuming that no loops or recursive functions need to be abstracted. Then, during each call to `PrecondSlice` (line 13 of Figure 3.4), we check if any value added to the slice's *live* set was defined in a synchronization-free loop L or recursive function R. If such an L or R is found, it must be abstracted.

How Do We Construct Abstractions? We construct a symbolic function $F_L(\overline{x}) = \overline{y}$, where $\overline{x}$ is the set of *live-ins* for loop L and $\overline{y}$ is the set of *live-outs*, where $\overline{x}$ and $\overline{y}$ can potentially be constructed with some form of summarization, such as the summarization algorithms proposed by Godefroid *et al.* [47, 51] (see Section 4.1.1 for a discussion). However, this is difficult in general since $\overline{x}$ and $\overline{y}$ can each include unboundedly many heap objects. Due to this difficulty, we currently construct each F_L manually. This process is interactive: we first run our algorithm from Section 3.3; if our algorithm finds a loop L that must be abstracted, it halts and reports L; we then produce a hand-written abstraction for L and re-run our algorithm.

During symbolic execution, we execute the abstraction F_L in place of the actual loop L. Each F_L should model the *terminating* behaviors of L. We require each F_L to terminate to ensure that our algorithm terminates as well. Of course, the actual loop L may not terminate, and we preserve that behavior—when the program is executed with our runtime system, we execute the actual loop L, not F_L.

Each F_L is allowed to be an *over-approximation* of loop L's terminating behaviors. This eases construction of F_L but adds potential to explore infeasible paths. Producing each F_L is usually not hard in practice, as the loops to abstract are often hidden behind natural abstraction boundaries. Continuing the above example, suppose the binary tree interface includes `TreeAdd` and `TreeDelete`. These appear difficult to abstract since they can mutate unboundedly many heap objects (*e.g.*, to rebalance the tree), but as long as all modifications and traversals are performed behind the `Tree*` interface, we can conservatively model `TreeAdd` and `TreeDelete` by simply generating a fresh symbolic value that represents the new root of the tree.

Although the above explanation is phrased in terms of loops, recursive functions can be abstracted in the same way.

3.5 Forming Efficient Bounded Epochs

So far we have assumed that in a given epoch, no thread executes beyond its *next* epoch marker. Why might this be inefficient? First, runtime performance is optimal when threads execute a *balanced* amount of work per epoch, but naïvely stopping at the next epoch marker can lead to imbalance. Second, epochs should be long enough so that ordering-dependent bugs, such as atomicity violations, are usually contained within a single epoch.

It is more efficient to allow each thread to bypass a finite number of epoch markers within each bounded epoch. Since epoch markers are placed in loops, we consider this is a form of *loop unrolling*. This optimization coordinates with our runtime system as follows: for each epoch marker bypassed by `ExecutePath`, we add a special node to the current happens-before schedule so that our runtime system will bypass that marker at runtime. We use the following heuristics to bypass epoch markers:

Minimum Epoch Length. A large body of prior work has made the empirical observation that most ordering-dependent bugs occur over a short execution window containing at most W instructions per thread. For example, the authors of AtomAid [75] estimate that W is "thousands of instructions" in the worst case, but "hundreds" in the common case. The authors of ColorSafe [74] estimate $W = 3000$. Other authors [25, 82] support this observation but do not give concrete estimates for W, although Burckhardt *et al.* [25] observe that many "hard" atomicity violations have the form `if(x) compute(x)`, where W spans the short window between the condition and the computation. These observations from prior work suggest the following simple heuristic: each thread should execute a minimum of W instructions per epoch—this ensures that epochs are large enough so that *most* concurrency bugs fall within either a single epoch, or two adjacent epochs.

Balanced Epoch Lengths. Each thread should execute approximately the same number of instructions per epoch. For example, suppose we are about to end an epoch with T_1 and T_2 stalled at epoch markers. If $len(T_1) > len(T_2) + k$, where $len(T_i)$ is the number of instructions executed by T_i in the current epoch and k is a heuristically-chosen constant, then we continue executing T_2 up to its next epoch marker.

3.6 Implementation

3.6.1 Symbolic Execution Engine

We implemented the above algorithms in a version of the Cloud9 [24] symbolic execution engine extended with the techniques described in Chapter 2. Cloud9 executes multithreaded C programs that use pthreads and compile to LLVM bitcode. To support unmodified C programs, Cloud9 includes hand-written symbolic models for the Linux system call layer and the pthreads library, and it models other C library functions by linking with an actual libc implementation (uClibc). We have instrumented Cloud9's pthreads library to dynamically capture a happens-before schedule during symbolic execution. We attempted to model the pthreads specification faithfully; for example, our model includes various error paths for pthreads functions, in addition to the common-case success paths.

Limitations. Our implementation has a few limitations that we consider minor but list for completeness: async signals, C++ libraries, and floating point arithmetic. First, we do not support asynchronous delivery of POSIX signals. This has not been a problem so far. Should it become an issue, we can support asynchronous delivery by buffering signals until epoch boundaries, similarly to OS-level systems for deterministic execution [14] and record-and-replay [63]—such a buffering scheme would eliminate the need to reason about a combinatorial explosion of possible signal delivery points.

Second, Cloud9 ships with a standard C library (uClibc [1]) but not a standard C++ library, and this limits our ability to run C++ programs. Third, our underlying theorem prover, STP [46], does not support floating-point arithmetic. Cloud9 makes progress through floating point arithmetic by concretizing values, which means the resulting path constraints will be incomplete for paths that branch on the result of a floating-point computation. This is often not an issue for our algorithm in practice, since many programs compute floating-point results but do not using floating-point values to decide when to synchronize. However, this does prevent us from analyzing some programs, as we discuss later in our evaluation (Section 3.8).

Challenges. The effectiveness of precondition slicing is heavily dependent on the presence of a good whole-program alias analysis. The critical operation is the `WritesLiveVar-`

`Between` check (Figure 3.3)—alias analysis imprecision can lead to the incorrect belief that a live variable was written, which results in an overly strong schedule precondition, which results in the exploration of redundant schedules.

Our implementation uses DSA [67], which, in whole-program mode, degrades to a field-sensitive Steensgaard (equality-based) analysis. Our experience suggests that an inclusion-based analysis is *vital*. The problem intensifies because we link with an entire C library—all pointer variables passed to library functions are effectively merged in the points-to graph. We unfortunately could not find a publicly available alias analysis for LLVM that is more powerful, so we hacked around this problem by dividing pointer variables into two classes: application code and library code. Variables in the later class are assumed to alias anything, while variables in the former class are analyzed with DSA.

3.6.2 Compiler Instrumentation and Runtime System

Our symbolic execution engine outputs a database of input-covering schedules that our runtime system follows faithfully. Recall from Section 3.3 that this database maps each epoch E $\in \mathcal{E}$ to an input-covering set Σ_E for E.

At a high-level, our runtime system is mostly straightforward. The global variable `currSchedule` contains the happens-before schedule for the currently executing epoch. At the beginning of the program, we compare the current inputs with the database of input constraints to select the initial schedule. Similarly, epoch markers are turned into barriers, and when all threads reach an epoch barrier, a single thread is selected (arbitrarily) to update `currSchedule` for the next epoch. Then, at each synchronization statement, the runtime system inspects the calling thread's current happens-before node, waits until all incoming happens-before dependencies are satisfied, and then advances to the next node.

A more detailed view is given in Figure 3.6. We map each epoch E to a *schedule selector function* F_E for each epoch. Schedules contain a list of happens-before nodes for each thread. There are three technical challenges: At each epoch barrier, how do we efficiently determine the next epoch id E? How do schedule selector functions check input constraints? And, how do we deal with dynamic inputs?

<u>**Global state**</u>

```
  struct ScheduleFragment {
    nextSelectorId: int
    schedule: map (threadId, list of H-B-Node)
  }

  selectors: map(int, (void)->ScheduleFragment*)
  currCallstacks: map(threadId, int)
  currSchedule: list of ScheduleFragment*
```

<u>**Schedule selection at epochs**</u>

```
  EpochBarrier() {
    isLast = barrier.arrive()
    if (isLast) {            // last thread?
      epochId = hash(sort(currCallstacks.values))
      currSchedule.clear()
      currSchedule.append(selectors[epochId]())
      barrier.release()
    } else {
      barrier.wait()
    }
  }
```

Figure 3.6: Key components of our runtime system.

Determining the Next Epoch. Each epoch id E is defined by a multiset of per-thread call stacks (recall Section 3.4.1.3). We instrument the program to record each thread's call stack in a globally-visible location. Then, the last thread to arrive at an epoch barrier can compute the next epoch id E by sorting this list of call stacks (note that a multiset can be represented by a sorted list). The sorting operation is made efficient by representing call stacks using hash values as in PCC [22]. The algorithm used by PCC has only probabilistic guarantees that each calling context is given a unique hash value, but since we know the complete set of epoch ids, we can ensure *a priori* that a unique hash value is computed for each epoch.

Schedule Selector Functions. When invoked, the selector F_E looks for a pair (I,S) $\in \Sigma_E$ such that constraint I matches the current input, then it return S. This is implemented by compiling Σ_E into a decision tree. Our current implementation selects each schedule as a deterministic function of the given input, though this could easily be changed to select schedules nondeterministically when multiple options are available.

Recall that an epoch's *input* can include the state of memory. The difficulty is that the choice of schedule can depend on thread-local variables, where thread-local variables include stack-allocated variables as well as statically-allocated variables declared with gcc's `__thread` attribute. Since F_E is executed by one thread only, how does it reason about state local to other threads? Our solution is to instrument the program to maintain a globally-visible shadow copy of each thread-local variable that is used in input constraints. In practice this is a very small percentage of all variables, as we demonstrate in Section 3.8.2. Note that we must also make shadow copies of variables that are needed to reach heap objects used in input constraints. For example, if a constraint depends on the value of `x->next->data`, where `x` is a thread-local variable, then we must maintain a shadow copy of `x` to ensure that the `data` field is globally reachable.

Supporting Dynamic Inputs. As hinted in Section 3.2.1, we represent schedules as a tree of *schedule fragments*. At the beginning of an epoch, each thread follows the initial fragment, represented in Figure 3.6 as `currSchedule[0]`. Each fragment f ends in one of three ways: at program exit, at an epoch boundary, or at a new input read by thread T. In the latter case, thread T invokes the selector function named by f`->nextSelectorId`, then appends the selected fragment to `currSchedule`. As other threads arrive at the end of fragment f, they must wait for T to select the next fragment before proceeding. We ignore inputs that are pruned by precondition slicing (Section 3.3.2), so updates to `currSchedule` occur only after the arrival of inputs that can affect synchronization.

Dynamic inputs introduce a further challenge, best illustrated by the following sequence of events:

```
1 EpochBarrier()
2 z += 5
3 ReadInput(&x)
4 if (x == z && y == w) { lock() }
```

The selector function invoked at line 3 will evaluate the term `x == `z_0`+5`, where z_0 is the value of `z` at the beginning of the epoch. This value has been lost due to the update at line 2, so we need to *snapshot* `z` at line 1. Note, however, that we do not necessarily need to snapshot `y` or `w`—the condition `y==w` does not depend on input `x`, so it can be lifted into the epoch's selector function that is invoked at line 1.

Chances for Further Optimization. Runtime system optimization has not been our focus. We see at least three potential improvements: (1) we can apply a transitive reduction [97] on each happens-before schedule to reduce cross-thread synchronization; (2) for each term evaluated by selector function F_E, we can memoize the value of that term as computed by F_E to avoid recomputation during actual program execution; and (3) we can parallelize F_E to avoid serializing F_E at each epoch boundary (this last proposed improvement is perhaps the most complex).

3.6.3 Verifying Deadlock Freedom

We already check for deadlocks during our search for input-covering schedules (see Figure 3.4, line 18). So, in a sense, we get deadlock checking for "free." Our algorithm either outputs a set of non-deadlocking schedules, in which case we are guaranteed to never deadlock at runtime, or its output will include at least one pair (I,S) where schedule S deadlocks, in which case we *may* deadlock at runtime. In the later case, we cannot prove that deadlock will actually occur at runtime because input constraint I may be infeasible (recall Section 3.3.3). In this way, our deadlock checker is imperfect. Currently, we manually inspect deadlocking schedules to determine if they are actually feasible, but we hope to use more sophisticated strategies for removing infeasible paths in future work to make these manual checks unnecessary.

3.7 Discussion of Guarantees

Given a program P, our schedule enumeration algorithm outputs a set of bounded epochs $\mathcal{E}$ along with a set of input-covering schedules Σ_E for each epoch $E \in \mathcal{E}$. Our schedule enumeration algorithm and runtime system combine to provide the following guarantees, which we state without proof:

Property 1 (Completeness of Σ_{E_0}). *Suppose execution begins from program entry with initial memory state M_0, where M_0 contains nothing except the program's inputs. If Σ_{E_0}*

is the set of input-covering schedules for E_0, the epoch at program entry, then for all valid M_0, there must exist a pair $(I, S) \in \Sigma_{E_0}$ such that M_0 satisfies constraint I.

Property 2 (Soundness and Completeness of $\mathcal{E}$ and all Σ_E)**.** *Suppose execution begins from a program context that corresponds to some epoch $E \in \mathcal{E}$, and suppose the initial memory state is M.*

Then, for all pairs $(I, S) \in \Sigma_E$ where M satisfies constraint I, if our runtime system forces execution to follow S, then either: (a) execution will encounter a data race; or (b) execution will follow schedule S without deviation. In case (b), schedule S must terminate at program exit, at a deadlock, or at some subsequent epoch $E' \in \mathcal{E}$. If schedule S terminates at epoch E', then execution must arrive at E' with a memory state M' such that there exists a pair $(I', S') \in \Sigma_{E'}$ where M' satisfies constraint I'.

Properties 1 and 2 establish that our system is both sound and complete for race free programs. By *sound*, we mean that for any epoch $E \in \mathcal{E}$ and any pair $(I, S) \in \Sigma_E$, it must be possible for execution to follow schedule S when given an appropriate initial memory state. By *complete*, we mean that, for all possible program inputs, execution will proceed through a (possibly nonterminating) sequence of epochs $E_0, E_1, E_2, \cdots$, where each E_i exists in $\mathcal{E}$, and as execution arrives at each epoch E_i, there must exist a schedule $S_i \in \Sigma_{E_i}$ such that execution can be constrained to S_i within that epoch. Soundness is established by Property 2, and completeness is established by Property 1 combined with inductive application of Property 2.

The important consequence of Properties 1 and 2 is that verification tools can reason soundly and completely even when they consider only those schedules contained in Σ. Of course, these properties hold *only when* the program's execution is constrained by our runtime system—when execution does *not* use our runtime system, Σ under-approximates the set of schedules that might be followed and our verification guarantees are voided. This is why we intend to use our runtime system in *all* executions of a given program.

Additionally, our system is subject to two categories of limitations:

Fundamental Assumptions. As stated in Section 3.1, our approach *fundamentally* assumes, first, that programs are data race free, and second, that programs have a bounded number of live threads at any moment. When the first assumption is broken, our schedule enumeration algorithm is unsound and execution can diverge from the expected schedule at runtime. When the second assumption is broken, our schedule enumeration algorithm will not terminate.

Limitations of our Implementation. As stated in Section 3.6.1, our implementation has limited support for async signals, C++ libraries, and floating point arithmetic. As stated in footnote 1 in Section 3.3.1, our implementation does not support recursive functions that synchronize. Properties 1 and 2 do not hold for programs that exceed these limitations. However, these limitations are specific to our implementation and are not fundamental to our approach.

Full proofs of Properties 1 and 2 are beyond the scope of this dissertation. Full proofs would require a model of execution, a model of the runtime constraint system, and either assuming correct or proving correct our slicing algorithm (based on precondition slicing, which was described without a formal proof of correctness [32]), our underlying symbolic execution engine [24], and our underlying SMT solver [46].

3.8 Evaluation

Our evaluation is organized in three parts. We start with a set of case studies (Section 3.8.1) that evaluate the effectiveness of our schedule enumeration algorithm on a range of applications. Our case studies include selections from the Splash2 and Parsec benchmark suites, as well as `pfscan`, a parallel implementation of `grep`. Along with each case study, we include a characterization of the effectiveness of our optimizations—this helps characterize the program analysis challenges inherent to enumerating input-covering schedules. We then evaluate our runtime system (Section 3.8.2). We end by evaluating how well our symbolic execution techniques described in the previous chapter aid the schedule enumeration algorithm described in this chapter (Section 3.8.3).

We ran all experiments on a 4-core 2.4 GHz Intel Xeon E5462 with 10GB RAM. Each core had 2-way hyper-threading enabled, resulting in a total of 8 hardware contexts. For each application, we marked all command-line parameters as input, with the exception of each application's "num threads" parameter, which we fix to values such as 2, 4, and 8 to reveal how our analysis and our runtime system scale with increasing thread counts. Capturing command-line inputs required a minor code change of about 10 lines per application. Other inputs derive from values returned by system calls, which include values read from files—these inputs are captured automatically. Two applications required additional minor code changes that we describe later (see Sections 3.8.1.2 and 3.8.1.4).

We attempted to analyze most programs that were analyzed by the related schedule memoization system PEREGRINE [35], but occasionally ran into limitations of our implementation (recall Section 3.6.1 and footnote 1). Specifically, we could not run: `barnes` and `ffm` from SPLASH2, which perform synchronization in recursive functions; `pbzip`, which uses C++ libraries; and `ocean`, `fluidanimate`, and `streamcluster`, which use floating-point arithmetic to control synchronization.

3.8.1 Case Studies

For each case study, we address the following major questions: Is a set of input-covering schedules enumerable in a reasonable amount of time? And if so, how large is $\mathcal{E}$, how large is each Σ_E, and how large is the total set of schedules (Σ)? We also attempt to characterize how many of those schedules are infeasible.

Overall results for our *fully optimized* algorithm are summarized in Tables 3.1 and 3.2.[2] In Table 3.1, Column 2 gives the maximum number of threads live at any given instant. This is a function of the application's "num threads" parameter, which we fix to 2, 4, and 8 for all benchmarks except `pfscan`, where we fix this parameter to 1, 2, and 3. Columns 3–9 summarize our algorithm's final output: Column 3 is the number of reachable epochs ($|\mathcal{E}|$); Columns 4–6 give statistics that summarize the number of schedules per epoch ($|\Sigma_E|$); and Columns 7–9 give statistics that summarize schedules across all epochs ($|\Sigma|$), including the total number of infeasible schedules and deadlocking schedules. For all but one program, we proved that Σ was deadlock-free. We determined the number of infeasible schedules through manual inspection of Σ. For `pfscan`, the schedules were too numerous for manual inspection, so we give a lower bound in Table 3.1.

[2] These tables contain data updated relative to our paper from OOPSLA 2013 [11], reflecting bug fixes that have been made since that original publication.

App	#thr	$\lvert\mathcal{E}\rvert$	$\lvert\Sigma_E\rvert$ min	max	avg	Summary of Schedules ($\lvert\Sigma\rvert$) Total	Infeasible	Deadlocked
blackscholes	3,5,9	1	1	1	1	1	0	0
swaptions	3,5,9	1	1	1	1	1	0	0
fft	2,4,8	1	2	2	2	2	0	0
lu*	2,4,8	2	1	2	1.5	3	0	0
radix*	2	3	1	2	1.1	6	1	1
radix*	4	3	1	4	1.8	9	4	4
radix*	8	3	1	16	3.2	22	17	17
pfscan*†	2	15	2	127	25.1	455	49+*unk.*	49
pfscan*†	3	47	2	1209	110.7	7297	411+*unk.*	411
pfscan*†	4	50+	2	3792	404.8	26000+	6000+*unk.*	6000+

Table 3.1: Evaluation of our input-covering schedules enumeration algorithm. This is the fully-optimized algorithm. Applications marked with * use a *"join on all threads"* optimization (see Section 3.8.1.2), and applications marked with † use manually-constructed input abstractions (recall Section 3.4.2, and see further discussion in Section 3.8.1.4).

App	#thr	Analysis Runtime
blackscholes	3,5,9	5 s, 6 s, 14 s
swaptions	3,5,9	4 s, 9 s, 65 s
fft	2,4,8	7 s, 306 s, 90 m
lu	2,4,8	6 s, 7 s, 11 s
radix	2	9 s
radix	4	10 s
radix	8	53 s
pfscan	2	24 s
pfscan	3	54 m
pfscan	4	10+ h (*dnf*)

Table 3.2: Runtime of our input-covering schedules enumeration algorithm. This is the fully-optimized algorithm. We set a time limit of 10 hours, and the one application that could not be analyzed within this time limit is marked *did-not-finish (dnf)*.

App	#thr	# epoch markers w/ §3.3.1	no §3.3.1
blackscholes	9	0	2
lu	8	0	2
radix	8	2	4
pfscan	3	3	7

Table 3.3: Cost of disabling epoch marker optimizations.

		no §3.4.1.1			no §3.4.1.3			no §3.4.1.4		
App	#thr	$\lvert\mathcal{E}\rvert$	$\lvert\Sigma\rvert$	time	$\lvert\mathcal{E}\rvert$	$\lvert\Sigma\rvert$	time	$\lvert\mathcal{E}\rvert$	$\lvert\Sigma\rvert$	time
blackscholes	9	1	***3***	6 s	—	—	—	—	—	—
lu	8	***1+***	∞	***dnf***	***5***	***9***	***22 s***	***4***	***7***	***18 s***
radix	8	3	***86***	53 s	***42***	***622***	***619 s***	—	—	—
pfscan	3	30	***3356***	***350 s***	***50***	***4772***	***405 s***	—	—	—

Table 3.4: Cost of disabling other optimizations. Compare ***bold*** values in columns labeled $|\mathcal{E}|$ and $|\Sigma|$ with Columns 3 and 7 in Table 3.1, respectively. Futher, compare ***bold*** values in "time" columns with values in Table 3.2. We mark columns with a dash (—) when the corresponding optimization has no effect.

Table 3.2 evaluates the runtime of our fully-optimized schedule enumeration algorithm. Column 2 gives the maximum number of threads live at any given instant, as in Table 3.1, and Column 3 gives our algorithm's runtime in seconds (s), minutes (m), hours (h), or *dnf* when our algorithm did not finish within 10 hours. Our algorithm completed in a reasonable period of time for most applications, with the the exception being `pfscan`, which we discuss in Section 3.8.1.4.

Optimizations. Tables 3.3 and 3.4 characterize the benefits of our optimizations over a representative subset of applications. We are particularly interested in how our optimizations affect the size of our algorithm's output ($|\mathcal{E}|$ and $|\Sigma|$) and our algorithm's runtime.

In Table 3.3, Columns 3 and 4 give the number of epoch markers added by `PlaceEpochMarkers`, both with and without the optimizations described in Section 3.3.1. Our optimizations reduce the number of epoch markers, often by 50%. Fewer epoch markers means fewer epochs, which means less combinatorial explosion and a more scalable algorithm. Hence, we consider these optimizations *essential*.

In Table 3.4, Columns 3–11 show the results of running our schedule enumeration algorithm with specific optimizations disabled. Optimizations are named by the section in which they are described. In each group of columns, $|\mathcal{E}|$ is the number of enumerated epochs (as in Column 3 of Table 3.1), $|\Sigma|$ is the total number of enumerated schedules (summed across all epochs, as in Column 7 of Table 3.1), and *time* is the analysis runtime (as in Table 3.2). Most of our optimizations are essential. For example, without the prefix schedules optimization enabled (Section 3.4.1.1), our algorithm explores an essentially unbounded number of failure paths in `lu` and never escapes the first epoch. (The number of failure paths could be bounded by applying input abstraction to two loops in `lu`, but the key point is that `lu` does *not* require input abstraction when the prefix schedules optimization is enabled.)

Further, after disabling the symbolic thread ids optimization (Section 3.4.1.3), we observed a significant increase in $|\Sigma|$, by as much as two orders of magnitude. We consider this optimization essential, as well. The optimization to avoid code duplication (Section 3.4.1.4) improved `lu` only. However, as described later, in Section 3.8.1.4, our algorithm enumerates redundant schedules in `pfscan` as a result of a form of code duplication that our code-duplication optimization could not eliminate.

We unfortunately could not *meaningfully* evaluate the effectiveness of our optimization to ignore library synchronization (Section 3.4.1.2). The root of the problem is that our points-to analysis treats library code very conservatively (recall Section 3.6.1): as a result, preconditioning is extremely imprecise when operating on library code, leading to an explosion in redundant schedules. We consider this an artifact of our implementation, and specifically an artifact of our choice of points-to analysis, rather than a fundamental property of our system, and we believe that the optimization from Section 3.4.1.2 could be meaningfully disabled if a stronger points-to analysis was available.

3.8.1.1 The Trivial Case: Fork-Join Parallelism

blackscholes (from PARSEC) uses fork-join parallelism with no other synchronization. It is so simple that we consider it the "hello world" of synchronization analysis. **swaptions** (also from PARSEC) is equally simple. Our algorithm easily infers that these applications need exactly *one* schedule for a given thread count.

3.8.1.2 Case Study: Barrier-Synchronized Parallelism

fft (from Splash2) uses fork-join parallelism with a static number of barriers. Our algorithm infers that `fft` needs just two schedules for a given thread count. The high analysis runtime is due to the presence of long sequences of conditionals that use division and modulo arithmetic in a way that our SMT solver (STP) finds challenging.

lu (from Splash2) synchronizes using a dynamic number of barriers. Our algorithm divides `lu`'s schedules into two epochs: one that begins at program entry (E_1), and one that begins within `lu`'s main parallel loop (E_2). We need just one schedule for epoch E_1 and two schedules for epoch E_2. In epoch E_2, one schedule traverses one lock-step iteration of the parallel loop, and the other exits the program.

`lu` introduces two program analysis challenges. First, we must infer that all threads execute the main parallel loop in lockstep. Failure to infer this fact results in infeasible schedules, as we will demonstrate when evaluating our symbolic execution techniques in Section 3.8.3.

Second, `lu` makes calls of the form `pthread_join(t[i])` that are deceptively difficult to analyze. We are unable to uniquely identify each `t[i]`, so we must analyze three paths per call: one in which `t[i]` is an invalid thread id, another in which `t[i]` refers to a thread that has already exited, and another in which `t[i]` refers to a thread that has not yet exited. In total, to join with N threads, we analyze 3^N paths, even though just *one* of those paths is feasible. We avoid this difficulty by replacing calls to `pthread_join` with a high-level *"join on all threads"* operation that is easy to analyze. Each application marked with an asterisk in Table 3.1 was modified to use this operation in place of `pthread_join`.

3.8.1.3 Case Study: Barriers and Semaphores

radix (from Splash2) is barrier-synchronized like `lu`, but with the addition of two parallel phases that use semaphores to coordinate a tree-based reduction. These semaphores present the major difficulty—as shown in Table 3.1, we explore a number of infeasible schedules.

The following example demonstrates the problem:

```
Thread 1          Thread 2
```

```
2 for (...) {       for (...) {
3   epochMarker()     epochMarker()
4   sem_wait(&s)      sem_post(&s)
5   ...               ...
```

`MakeStateForEpoch` (Section 3.3.3) cannot prove that `s.count==0` at the beginning of the epoch. (In the actual code, this is difficult because each `&s` is selected from an array.) As a result, we explore an infeasible schedule in which T_1 does *not* block at line 4 because it assumes that `s.count>0`. This schedule incorrectly synchronizes T_1 and T_2, which can lead to the (incorrect) conclusion that the program contains data races.

It is actually quite easy to *prove* that the above schedule is infeasible. Our insight is to exploit Σ, which pairs each schedule with an input constraint `I`. For the above schedule, `I` is `s.count>0`. Let E be the epoch containing that schedule. Our job is to show that each schedule $\mathrm{S}_i \in \Sigma$ that terminates at epoch E always terminates with $\neg$`I` resolving to true. This is easy to show using rely-guarantee reasoning: we add $\neg$`(s.count>0)` as an *assertion* to the end of each S_i; we add $\neg$`(s.count>0)` as an *assumption* to the beginning of epoch E; and then we symbolically execute each epoch in $\mathcal{E}$ to ensure that the assertions are always satisfied. Note that the assumption is necessary because epoch E contains a schedule that "loops back" to itself.

It would be possible to automatically discharge the necessary verification conditions. We have not implemented this feature, but we have applied this approach to `radix` by manually annotating the program with assumption and assertion annotations to drive the verification procedure. We verified that the 17 schedules listed as "infeasible" in Table 3.1 are truly infeasible.

Why were we able to prove infeasibility so easily both in the above example and for `radix`'s 17 infeasible schedules? The reason is that each input constraint `I` happens to be a function of synchronization state *only.* If some `I` was instead a function of arbitrary program state, we would need to consider all *paths* that terminate at epoch E, rather than just all *schedules.* The interesting novelty in our proof is that, since we had a small number of schedules to consider, we could reason about each schedule in isolation by reusing *sequential* rely-guarantee reasoning techniques.

3.8.1.4 Case Study: Task Queues and Locks

pfscan uses task parallelism with one producer thread and multiple worker threads, and it uses locks to guard shared data. The queue is implemented with locks and condition variables. `pfscan` has the following high-level structure:

```
1 Producer           Consumers
2 for (f in files)   while (dequeue(&f))
3   enqueue(f)         scanfile(f)
```

`scanfile` implements string matching. We had to abstract one loop (in `scanfile`) using the technique described in Section 3.4.2. This loop computes the next matching substring: the loop's live-ins include a string buffer and a current position, and the loop's live-out is the position of the next match. With 4 threads, were unable to enumerate a complete set of input-covering schedules within a ten hour time limit.

Interestingly, the prefix schedules optimization (no Section 3.4.1.1 in Table 3.4) does not help `pfscan` much at all. The reason is that `pfscan` acquires a lock on almost every failure path to perform logging. In fact, the majority of schedules enumerated for `pfscan` are needed to handle these failure paths: with 4 threads, at least one thread executed a failure path on about two-thirds of all enumerated schedules. Since all failure paths acquire the *same* lock, they could conceivably be merged into one schedule—this is an interesting direction for future research.

For `pfscan`, our algorithm produces a set Σ that includes deadlocking schedules. These deadlocks are all infeasible. The deadlocks include two scenarios: (1) the producer believes the queue is full while the consumers have already exited, and (2) the consumers believe the producer has exited without first setting the "done" flag. We enumerate these false deadlocks because our implementation of `MakeStateForEpoch` is not powerful enough to produce constraints that precisely relate the queue's `capacity`, `count`, and `done` fields.

We also explore redundant schedules that arise from code duplication. Recall from Section 3.4.1.4 that compilers transform while loops to an if-then-do-while form. This transformation duplicates the `dequeue` call made by each consumer thread in line 2 of the above code snippet. If this transformation could be disabled, we would reduce the number of enumerated schedules by about half.

App	IPE	#var instrum	DTree Size max	DTree Size avg	Norm. Exec Time 2thr	Norm. Exec Time 4thr	Norm. Exec Time 8thr
blackscholes	all	0	0	0	1.0	1.0	1.0
fft	all	0	1	1	1.0	1.0	1.0
lu	200M	1	4	1.75	1.0	1.0	1.0
radix	1B	4	6	2.95	1.0	1.05	1.05
pfscan	6K	7	24	2.2	1.6	—	—

Table 3.5: Runtime system characterization. IPE is *avg. instructions per epoch*, and LI is *# local variables instrumented.*

3.8.2 Runtime System Characterization

Table 3.5 characterizes our runtime system in three ways, as described below. All numbers in Table 3.5 are based on executions of the final *instrumented* program which is linked with our custom runtime system. These executions are constrained to the input-covering schedules summarized in Table 3.1, and, except where otherwise mentioned, the "num threads" was set to 4 for all benchmarks except `pfscan`, where it was set to 2 (resulting in 3 live threads).

In Table 3.5, Column 2 reports the average number of instructions executed per thread in a single epoch (IPE). We counted instructions by instrumenting LLVM bytecode, so the actual number of x86 instructions executed may differ slightly. As discussed in Section 3.5, IPE should ideally be large enough to span most ordering bugs. Our average IPE is well over the window of 3K instructions that was suggested by Lucia et al. in [74]. Although the averages are generally much higher than 3K, we noticed some variability. For example, IPE for `radix` fluctuated between 30K instructions and 1 billion instructions, depending on which of two alternating phases was being executed.

Columns 3–5 characterize the amount of work performed to compute a new schedule. Column 3 states the number of local variables instrumented to maintain shadow copies, and Columns 4 and 5 state the maximum and average number of arithmetic and boolean operators used by schedule selector decision trees (recall Section 3.6.2).

Columns 6–8 characterize our runtime overheads. Each column states the execution time of the final instrumented program (linked with our runtime system) normalized to

Program	thr	$\|\Sigma\|$ / AnalysisRunningTime *Full*	*-RD*	*-SI*	*N*
fft	2	2/ 9s	2/12s	2/ 10s	2/ 11s
lu	4	3/ 6s	23/14s	1550/396s	1976/202s
pfscan	2	455/24s	455/28s	2245/ 78s	2273/ 80s

Table 3.6: Symbolic execution characterization. *Full*, *-RD*, *-SI*, and *N* are as described in Section 2.9.

nondeterministic execution with the same number of threads. A value of 2.0 means "twice as long." For benchmark applications, we used standard benchmark workloads, and for `pfscan`, we performed a search in a directory containing 50 files. For barrier-synchronized applications, overhead is minimal—the programs are already designed to execute in a bulk-synchronous fashion. For `pfscan`, we were unable to measure how well our runtime scales with increasing threads, since we were unable to enumerate input-covering schedules for more threads within our time limit.

3.8.3 Symbolic Execution Characterization

Recall from Section 3.3.3 that our schedule enumeration algorithm uses the same symbolic execution techniques described in Chapter 2 of this dissertation. In particular, `MakeStateForEpoch` uses a context-specific dataflow analysis to construct each symbolic state, and `ExecutePath` performs symbolic execution using the algorithm from Figure 2.15.

We evaluated how well our schedule enumeration algorithm benefits from those symbolic execution techniques. Specifically, we ran our schedule enumeration algorithm using our symbolic execution techniques at various optimization levels. These results are shown in Table 3.6. For each optimization level, we give the number of schedules enumerated ($|\Sigma|$) and the total analysis runtime. We expect to enumerate more schedules at lower optimization levels (due to more infeasible paths).

Column 3 restates results for the fully optimized symbolic execution—this duplicates Column 7 of Table 3.1 and Column 3 of Table 3.2. Columns 4–6 show results when some optimizations are disabled. As in Section 2.9, *-RD* disables reaching definitions, *-SI* disables synchronization invariants, and N uses a *naïve* approach. Similarly to the infeasible paths evaluation in Section 2.9, if a schedule is enumerated with the *naïve* approach, but not the fully-optimized approach, then it must be an *infeasible* schedule. The results show, again, that our symbolic execution techniques are essential: the *naïve* approach suffers from slower algorithm runtimes and more infeasible schedules.

Chapter 4

RELATED WORK

We discuss related work in four sections. The first section summarizes prior work on symbolic execution, giving special attention to techniques that our work exploits or extends (Section 4.1). The next three sections summarize prior work related to input-covering schedules, including other work on constraining multithreaded execution (Section 4.2), work on summarizing multithreaded schedules (Section 4.3), and closely-related work on verifying multithreaded programs (Section 4.4).

4.1 Symbolic Execution

There has been much prior work on symbolic execution—too much to be completely surveyed here. In this section, we summarize prior work that is most relevant to our approaches. First, the goal of starting symbolic execution from arbitrary program contexts is to avoid path explosion, so we start by surveying other ways to avoid path explosion (Section 4.1.1). *All* of these approaches are complementary to each other and to our new techniques. Second, a major component of our symbolic execution algorithm is a novel model of the heap, so we survey symbolic heap models used by prior work (Section 4.1.2). We are not aware of any prior work that integrates a dataflow analysis with symbolic execution to the extent of our semantics as described in Chapter 2.

4.1.1 Approaches to Avoiding Path Explosion

Search heuristics are simple and effective. For example, one popular heuristic is to bias symbolic execution towards paths that are likely to by "most different" from paths that were already explored. This heuristic relies on a notion of "coverage", and many such notions have been explored: one approach is to measure coverage of individual branch edges or program statements [27]; another approach is to measure coverage of path subsequences [69]; a third approach is to avoid visiting the same symbolic state more than once, if said state happens to be reachable via multiple paths [60]; and a fourth approach extends the third approach by using future read and write sets to more precisely determine when two symbolic states are equivalent [23]. For multithreaded programs, a popular heuristic is to search the schedule space using iterative context bounding [24, 78].

Another common approach is *summarization*, in which a fragment of code is replaced by a symbolic summary of its behavior. This approach has been applied to summarize loop iterations [51] and procedure bodies [6, 47, 84, 93]. Some systems construct summarizes using an automatic analysis [47, 51, 84], while others rely on programmers to encode summaries manually [6, 93].

For multithreaded programs specifically, various *partial-order reductions* have been proposed [31, 45, 59]. The idea is to recognize, proactively, that some not-yet-explored schedule S describes the same partial order as some other previously-explored schedule S′—since both schedules have the same partial order, execution will produce the same result in both cases and schedule S can be pruned.

A final technique is *path merging* [52]. Path merging algorithms execute all paths connecting control-flow points A and B, then merge the resulting states at B. This reduces the size of the search space by a multiplicative factor. Specifically, if there are n paths connecting points A and B, and m paths following B, the search space reduces to $n + m$ paths with path merging compared to $n \cdot m$ paths without. However, when path merging is applied indiscriminately, the merged states become increasingly difficult to analyze. In practice, heuristics are needed to determine whether path merging will be more profitable than costly [61].

Our symbolic semantics (Chapter 2) uses a dataflow analysis to construct the initial symbolic state. Path merging might be considered a replacement for this analysis, but such a system would either suffer from path explosion (if all paths between program entry and the initial context are enumerated) or unsoundness (if some sample of those paths are enumerated). Prior work on path merging observed that the presence of large disjunctions creates SMT queries that current solvers find difficult to solve [52]—this observation motivated our use of *must*-reach definitions, which generate few disjunctions, rather than *may*-reach definitions, which generate many disjunctions (recall Section 2.3.3).

Comparison with Input-Covering Schedules Enumeration. In developing our algorithm to enumerate input-covering schedules, we faced three main technical challenges: infeasible paths (Section 3.3.3), redundant schedules (Section 3.4.1), and unbounded loops (Section 3.2 and Section 3.4.2). Each challenge represents a specific instance of symbolic execution's path explosion problem. For example, our optimizations to avoid schedule redundancies are reminiscent of the partial-order reductions described above, though that prior work identifies schedule redundancies given a fixed input, while we identify redundancies across inputs (cf. Section 3.4.1.1).

Further, in Section 3.4.2 we observed that some form of input abstraction is necessary to achieve good scalability of symbolic execution. Other authors have made the same observation, most notably Anand *et al.* [6] and Godefroid [48]. Anand *et al.* [6] propose using manually-written abstractions (as we do), and they propose a methodology for writing those abstractions. Such a methodology could be applied in our context.

Finally, one can view our use of bounded epochs as a form of path merging, where epoch boundaries represent path merge-points. Classical path merging algorithms execute all paths connecting two control-flow points A and B, then merge the resulting states at B. This is made feasible by keeping the distance between A and B short. Our algorithm does not execute all paths within each epoch, making it more challenging to construct initial states for middle-of-program epochs, so we rely on the symbolic execution techniques described in Chapter 2.

4.1.2 Symbolic Models of the Heap

A key property of any static analysis is its heap model. In the C language, each pointer has two components: a *base*, which specifies the base address of a heap object, and an *offset*, which specifies the specific interior byte of an object being pointed to. We survey heap models in two dimensions.

How powerful is the heap model? All symbolic heap models support symbolic pointer *offsets*, which allows reasoning about terms like `x[i]` where `x` is a known (concrete) location but `i` has an unknown (symbolic) value. A key differentiating factor is how completely a model supports symbolic *base* locations (`x` in the prior example). Many systems support execution from program entry *only*, enabling the assumption that memory objects always exist at concrete *base* locations. This approach is taken by Dart [49] and Sage [43, 50].

Other systems support symbolic *base* locations to some degree. Cute [90] performs symbolic unit testing of C programs. Each unit test analyzes a single function with unknown (symbolic) values for arguments, where the function's arguments can include pointers. However, Cute has very limited support for aliasing—two symbolic pointers `p1` and `p2` are allowed to alias only if the program contains an explicit comparison between the two pointers, such as a statement `if(p1==p2){...}`. Java Path Finder (JPF) performs unit testing of Java programs and supports symbolic *base* locations with arbitrary aliasing [60]. JPF supports aliasing by aggressively forking symbolic execution to explore all concrete heap graphs. For example, given two symbolic references `p1` and `p2`, JPF forks into two symbolic states: one in which `p1` and `p2` point to different concrete objects (`p1!=p2`), and another in which `p1` and `p2` point to the same concrete object (`p1==p2`). The number of forked symbolic states grows exponentially with the number of symbolic pointers, making this approach expensive in practice—in fact, we originally implemented this approach and found it unacceptably slow. Pex [92], klee [27], and `bbr` [29] all support symbolic *base* locations and can encode multiple heap locations in a single symbolic state, with varying degrees of efficiency, as described shortly.

How is the heap model implemented? A natural approach, taken by all modern analyses based on SMT solvers, is to represent the heap using the theory of arrays [46]. Reads and writes in the theory of arrays correspond directly to loads and stores of memory. A differentiating factor is *how many* arrays are used to represent the heap. Program verifiers often use just one array for the entire heap. For example, Havoc represents the heap as a simple mapping from integers to integers [28]. Other program verifiers follow Burtstall's memory model, which separates fields by name in a two-dimensional array [26]—this has been shown to be more efficient than the single-array approach [21, 87]. Pex follows the two-dimensional approach but adds constraints to disallow aliasing between objects of different types—this is sound because Pex operates over a type-safe bytecode [92].

In contrast, some systems use a *separate* array for each memory object. The authors of KLEE observed that symbolic execution engines often send many small queries to the SMT solver that can be resolved efficiently using caching [27]. For example, KLEE executes one solver query at every conditional branch statement. This contrasts with program verifiers, which summarize each program with a single large expression that is sent to an SMT solver just once. KLEE's key insight is that, by assigning each memory object its own array, symbolic expressions can avoid mentioning irrelevant parts of the heap, making caching more effective. To clarify, consider the following example:

```
if (z) { *y = 5; }
if (*x) ...
```

Two paths reach the statement `if(*x)`. If we can prove that the value of `*x` is the same on both paths, we can cache the result of *isSat*(`*x`) for reuse on the second path. KLEE and Cloud9 both implement this optimization, and Cadar *et al.* showed that this sort of caching optimization is vital for symbolic execution scalability [27]. Proving that `*x` has the same value on both paths is difficult when using a single global array for memory, as in this case, `*x` is $\mathit{read}(\mathit{write}(\mathit{global}, y_{\mathit{off}}, 5), x_{\mathit{off}})$ on one path, but $\mathit{read}(\mathit{global}, x_{\mathit{off}})$ on the other path. However, proving that `*x` has the same value on both paths can be trivial when the objects pointed to by `x` and `y` are represented with different symbolic arrays, A_x and A_y, as in this case, `*x` will be $\mathit{read}(A_x, 0)$ on both paths. Hence, to enable this caching optimization, our symbolic semantics (Section 2.4) assigns separates symbolic arrays to `x` and `y` when a points-to analysis can prove that `x!=y`.

`bbr` also uses a separate array per object [29]. However, KLEE and `bbr` use different approaches to resolve aliasing. To dereference a symbolic pointer `p`, KLEE opts to fork for each possible concrete *base* location, much like JPF, although KLEE performs this forking more lazily. In contrast, `bbr` uses guarded expressions to encode multiple concrete heap graphs into a single symbolic state. This approach is more efficient than KLEE's approach because it results in less forking, but results in more complexity. Dillig *et al.* [39] describe a heap model very similar to the one described by `bbr`, except that Dillig *et al.* developed their model in the context of a flow-sensitive, context-sensitive analysis rather than a path-sensitive symbolic execution. Our approach is most similar to that of `bbr` and Dillig *et al.*

4.2 Constrained Execution

Other systems have proposed ways to *constrain* multithreaded execution. Each of these systems has the same high-level goal as our work on input-covering schedules: to reduce the space of possible thread schedules. In this section, we contrast prior work in this area with our input-covering schedules based approach.

Schedule Memoization. The most closely related systems are TERN [34] and PEREGRINE [35]. These systems operate similarly to each other: First, they select some set of inputs which will be used to *test* program P. Second, they run P with each test input and record an execution trace. Third, they replay each execution trace in a symbolic execution engine to extract a schedule, S, and an approximate *weakest precondition*, I, where I is a constraint that describes a set of inputs that must be executable under schedule S. The final output is a database containing memoized pairs (I,S). At runtime, given input i, both systems search for a pair (I,S) such that i satisfies constraint I—if such a pair is found, execution is constrained to S, and otherwise, execution is unconstrained.

TERN computes weakest preconditions naïvely based on user annotations. PEREGRINE computes weakest preconditions with an algorithm based on *precondition slicing* [32], similarly to our work (Section 3.3.2). Both TERN and PEREGRINE have limited support for bug avoidance. Given an execution trace for input i, they run a simple race detector to find data races on that trace. If any races are found, the memoized schedule is discarded.

Our system generalizes ideas introduced by TERN and PEREGRINE. Specifically, TERN and PEREGRINE memoize schedules from a few *tested* inputs, so they provide best-effort schedule memoization only, while our system enumerates a *complete* input-covering set. Computing input-covering sets is a more difficult analysis problem, since it introduces the need to reason about all possible program behaviors, while TERN and PEREGRINE reason about behaviors on a few selected inputs only. The advantage of our approach over best-effort schedule memoization is clear: given set of input-covering that has been thoroughly tested, we can constrain execution in a way that *always* avoids executing untested schedules.

At a technical level, there are a few additional similarities. First, our notion of bounded epochs is related to TERN's idea of *windowing*, which handles a specific kind of unboundedness—event loops in server programs. TERN's windowing requires programmer annotations, but our system introduces epoch boundaries automatically.

Second, our system uses a weakest precondition computation similar PEREGRINE's. The main difference is that PEREGRINE's algorithm does *not* assume data race freedom. Instead, PEREGRINE uses a static may-race analysis to find memory access pairs that may-race and then adds a happens-before edge to the schedule for each such pair. Adopting this approach in our setting would result in a more complex analysis and more symbolic path explosion due to the need to consider many possible may-race pairs. Note that path explosion is not a problem in PEREGRINE's setting, where slicing is used to compute an input constraint for tested paths *only*, but not to select more symbolic paths.

Deterministic Execution. Recently there has been a flurry of research on deterministic execution [8, 10, 14, 15, 37, 38, 53, 70, 72, 81]. These systems constrain execution so that the resulting thread schedule is always a deterministic function of the program's input. Systems that enforce deterministic execution can be broadly classified in two categories. First, systems like Kendo [81] enforce a deterministic order on synchronization only, resulting in *weak determinism.* The Kendo algorithm can be implemented very efficiently in software, but it provides few guarantees for programs with data races—such programs may execute nondeterministically. Second, systems like DMP [37] enforce a deterministic order on all memory accesses, resulting in *strong determinism.* Strong determinism is attractive because it guarantees determinism even in the presence of data races. Unfortunately, strong determinism cannot be acheived on commodity hardware without imposing prohibitive overhead (empirical evaluations have observed overheads of up to 10$\times$ [8, 10, 14, 70]).

Our runtime system selects schedules deterministically for each input (Section 3.6.2). Since we assume data race freedom, we provide *weak determinism* as in Kendo. However, our primary goal is not determinism *per se*—we could just as easily map each input to schedules in Σ and randomly select from those schedules at runtime.[1] This added flexibility increases schedule diversity, which has potential benefits for security, fault-tolerance, and performance [12].

Most importantly, our approach *significantly* improves the testability and verifiability of multithreaded programs in comparison to deterministic execution. One of the original arguments in favor of deterministic execution was improved testability [37]. However, this claim is largely unproven and has been called into question [12, 98]. We summarize that counter-argument below:

Consider the problem of finding a concurrency bug, such as an atomicty violation, in some program P. With conventional nondeterministic execution, the bug will manifest under some set of triggering inputs $\mathcal{T}$ and set of schedules $\mathcal{S}$. With deterministic execution, the bug will manifest under inputs $\mathcal{T}' = \{i \mid i \in \mathcal{T} \wedge Det(i) \in \mathcal{S}\}$, where *Det* is the deterministic scheduler function. Observe that deterministic execution can actually *hide* the bug—in the worst case, *Det* produces a different schedule for each input i and the bug will manifest on a specific input only. Finding a specific input $i \in \mathcal{T}'$ is a very difficult problem. Further, with conventional nondeterminism, it is often the case that bugs manifest under many inputs and schedules [25, 77], meaning that $|\mathcal{T}|$ and $|\mathcal{S}|$ can be reasonably large in practice. Hence, it is not clear that concurrency bugs are any easier to find when execution is deterministic, than nondeterministic.

In contrast, concurrency bugs are *significantly* easier to find when execution is constrained to a small set of input-covering schedules. For a simple illustration, suppose the set of bug-triggering inputs $\mathcal{T}$ includes *all* possible inputs. With deterministic execution, we might need to test every input $i \in \mathcal{T}$ to find a manifestation of the bug. With our system, we need to test just one input for each pair (I,S) $\in \Sigma$.[2] When $|\Sigma|$ is small, it is obviously much simpler to test the program just $|\Sigma|$ times rather than once for every possible input.

Constraining Pairs of Accesses. Whereas schedule memoization and deterministic

[1] To give a concrete input i multiple mappings in Σ, we can add two pairs (I,S) and (I',S') to Σ such that input i satisfies both constraint I and constraint I'.

[2] Of course, we may need to test more than one input for each pair (I,S) $\in \Sigma$ when $\mathcal{T}$ does not include all possible inputs. The example described here is a "best case".

execution both constrain entire schedules, other approaches constrain specific instruction pairs only. One approach is to enumerate *predecessor sets* (PSets) for each memory access instruction [99]. The PSet for instruction *op*, named by PSet(*op*), is the set of remote instructions that *op* is allowed to depend on. For example, if *op* is a load instruction, then execution should be constrained so that *op* always reads a value written either (a) by the current thread, or (b) by a remote thread via one of the store instructions in PSet(*op*). Yu *et al.* proposed two hardware architectures for enforcing PSets: one that used delays with occasional checkpoint/rollback [99], and another that used transactional memory [100]. To reduce the chance some PSet contains a buggy interleaving, Yu *et al.* derive PSets from successful executions of test suites.

PSets define a *whitelist* of thread interleavings. The dual approach, taken by Aviso [73], is to define a *blacklist* of interleavings. Aviso derives its blacklist from interleavings that are known to be buggy, and it uses similar delay-based techniques to enforce interleaving constraints.

Our approach has a significant advantage over both PSets and Aviso: completeness. The system proposed by Yu *et al.*, cannot prove that the PSets collected during testing are sufficient to enable execution of all program inputs, so their hardware must necessarily resort to unconstrained execution in some cases. In Aviso, interleavings are constrained only after a bug is found, so unlike Yu *et al.*'s system, Aviso cannot help avoid bugs that have not been located. Further, Aviso's delay-based implementation gives only probabilistic guarantees of bug avoidance. In contrast, our system enumerates a complete input-covering set of schedules—so it does not suffer from the incompleteness of PSets—and our deployed system executes tested schedules only—so it does not suffer from the blindness of Aviso.

Automatically Avoiding Concurrency Bugs. Other systems take bug-specific approaches to constrained execution. For example, Atom-Aid [75] reduces the likelihood of triggering atomicity violations by executing large chunks of code transactionally. ISOLATOR [86] and ToleRace [88] use data replication to ensure isolation of critical sections, which avoids some data races. Dimmunix [58] and Communix [57] provide automated deadlock immunity for Java programs.

These systems are effective and usually have low overhead. However, their bug-specific nature limits their usefulness. It is difficult to imagine deploying many of these systems simultaneously. Further, some approaches (notably, Atom-Aid) are probabilistic—they reduce, but do not eliminate, the chance of encounting a buggy schedule. In contrast, through thorough testing of each schedule in the input-covering set Σ, our system provides bug avoidance for a range of bugs by constraining execution to tested schedules only.

4.3 Constructing Abstract Schedule Graphs

This section summarizes algorithms for constructing abstract *schedule graphs*, where a schedule graph abstracts a multithreaded program's set of possible runtime schedules much in the same way that a control-flow graph abstracts a single-threaded program's set of possible runtime paths.

Barrier Matching. In certain restricted cases, if a program is composed of barriers and no other kind of synchronization, it is possible to construct a set of input-covering schedules for the program using relatively simple algorithms. Jeremiassen and Eggers made an initial attempt at this problem [56]. Aiken and Gay gave a nice solution comprised of a simple bottom-up traversal over the syntax tree [5]. Zhang and Duesterwald extended Aiken's algorithm to support more kinds of loops and conditional statements [102]. Conceptually, these algorithms output a *barrier sequence graph* in which nodes are barrier operations and edges represent the happens-before relation. The graph may have cycles to denote loops. A given barrier graph can be viewed as a precise, compact representation of a set of input-covering schedules.

Although these algorithms give precise output, their domain is limited: they cannot reason about certain nontrivial loops or conditional statements, especially loops involving linked structures; they assume that barriers apply to all active threads, not some potentially input-dependent subset of threads; and, of course, they cannot produce input-covering schedules for programs that use synchronization other than barriers. In contrast, our approach supports any kind of synchronization operation that can be described with a happens-before relation, and further, our approach exploits the full power of symbolic execution to reason about a range of complex path conditions.

May-Happen-In-Parallel Analysis. This analysis is focused primarily on synchronization operations that can be classified as either a `notify` or a `wait`, such as explicit notify/wait operations on condition variables or exit/join operations on threads. The idea is to match each `notify` statement N with the set of `wait` statements $\mathrm{W_N}$ that *may* be notified by N. Given these mappings, we can construct a *concurrent flow graph* that combines each thread-local control flow graph with cross-thread edges that connect N with $\mathrm{W_N}$. The precise form of this graph varies by algorithm, and many algorithms have been proposed [4, 9, 40, 79, 80].

We can view the output of may-happen-in-parallel (MHP) analysis as an approximation of all possible happens-before graphs. So, in a sense, MHP analysis computes an approximate set of input-covering schedules. However, MHP analyses are path-insenstive, which makes them cheap to compute but *extremely* approximate relative to our fully path-sensitive algorithm for enumerating input-convering schedules.

4.4 Verifying Multithreaded Programs by Reduction to Sequential Programs

There has been much prior work on the verification of multithreaded programs—too much to survey here. An emerging idea is to analyze a multithreaded program by first reducing it to an equivalent sequential program. Prior techniques in this space are *incomplete*, in the sense that they do not analyze all possible schedules—in order to make the transformation from multithreaded program to sequential program feasible, they must pick some subset of possible schedules to seed the transformation. Two approaches have been proposed:

Reduction Under a Preemption Bound. A schedule is *preemption bounded* to depth k if the schedule includes no more than k preemmptions. Qadeer and Wu showed how to reduce a multithreaded program into a sequential program given a fixed k [85]. Subsequent authors have developed more advanced transformations [62, 64]. It has been shown empirically that many concurrency bugs can be found with $k \leq 2$, making this approach practical. For example, Qadeer and Wu used this technique to find data race bugs in Windows device drivers.

Reduction Given a Specific Schedule. An alternate approach, known as *schedule specialization*, considers one schedule at a time. The algorithm given by Wu *et al.* [96] takes as input a multithreaded program P along with a schedule S, represented as a total order of synchronization operations, and then outputs a program P′ specialized to schedule S. The specialized program P′ converts cross-thread use-def relations that are fixed by the given schedule into explicit communication. In experiments performed by Wu *et al.*, schedule specialization reduced the false positive rate of a static race detector by 69%.

Applicability to Input-Covering Schedules. Both of the above reductions are *incomplete* in practice—preemption bounding is incomplete unless $k \approx \infty$, and schedule specialization is incomplete unless all schedules are available. By constraining execution to a small set of input-covering schedules, our approach can make these promising reductions *complete*. Specifically, the approach to schedule specialization taken by Wu *et al.* [96] can be *directly* applied to our system by producing a specialized program for each schedule in Σ.

Chapter 5

CONCLUSIONS AND FUTURE OPPORTUNITIES

This dissertation described two ways to reason about multithreaded programs in the face of an enormous thread interleaving space. Our first approach is to symbolically execute small fragments of a program in isolation, rather than the entire program at once, by jumping directly to program contexts of interests. Our second approach is to constrain execution to small sets of input-covering schedules, avoiding that enormous interleaving space entirely. In developing these approaches, we introduced two new research problems:

The "symbolic execution from arbitrary contexts" problem: Given an initial program context, which we define to be a set of threads and their program counters, how do we efficiently perform symbolic execution starting from that context while soundly accounting for all possible concrete initial states? We proposed a solution to this problem in Chapter 2.

The "input-covering schedules" problem: What is the most efficient way to find and exploit sets of input-covering schedules? We proposed a solution to this problem in Chapter 3. As defined in that chapter, we say that a set of schedules Σ is input-covering for a given program P if, for each possible input i, there exists some schedule S $\in \Sigma$ such that, when program P is given input i, P's execution can be constrained to S and still produce a semantically valid result.

5.1 Summary of Conclusions

Broadly, we draw the following conclusions:

First, when solving the "symbolic execution from arbitrary contexts" problem, we found it profitable to integrate dataflow analyses with symbolic execution. Specifically, our empirical evaluations in Section 2.9 and Section 3.8.3 showed that two classes of dataflow analyses are particularly profitable: *reaching definitions*, to summarize the state of memory in a general way, and *synchronization invariants*, such as locksets, to summarize the state of synchronization objects in a specific way. In broader terms, we believe that practical solutions to the "symbolic execution from arbitrary contexts" problem *must* construct the initial symbolic state using a scalable analysis of some sort, and we have shown that scalable dataflow analyses can be a good fit.

Second, when solving the "input-covering schedules" problem, we found it necessary to enumerate input-covering schedules for bounded-length fragments of the program, rather than for the program as a whole. This strategy ensures that each input-covering set has a bounded size, preventing unbounded combinatorial explosion. In our system, such bounded-length fragments are called *bounded epochs*. In fact, the idea to analyze small fragments of a program in isolation, rather than analyzing the entire program as a whole, is a central concept in this dissertation that plays a key role in the core technical contributions of Chapters 2 and 3.

Third, we demonstrated the feasibility of building an end-to-end system for exploiting input-covering schedules. We implemented an algorithm to enumerate sets of input-covering schedules, we implemented a runtime system to constrain execution to those schedules, and we built a simple deadlock checker to look for deadlocks in those schedules. Our empirical evaluation in Section 3.8 demonstrated that it is possible to enumerate a complete set of deadlock-free input-covering schedules for at least some realistic programs.

5.2 Limitations

Our solutions to the above problems are not perfect. We summarize the most important limitations below:

Related to Symbolic Execution. We trade *scalability* for *precision*. A perfect solution would summarize all paths from program entry up to the initial context to compute a very precise initial state. Unfortunately, we believe this is not tractable in practice, so our approach is to compute an over-approximation of the initial state. This results in an analysis that is scalable but not fully precise. Hence, unlike classic approaches to symbolic execution, which are fully precise because they explore feasible paths only, our approach is imprecise and can explore *infeasible paths*—this can lead to *false positives* in verification tools, testing tools, and other analyses that build on our symbolic execution techniques.

Related to Input-Covering Schedules. Our approach to enumerating input-covering schedules assumes *data race freedom*. We make this assumption to simplify our analysis. However, when this assumption is broken, we do not compute a true set of input-covering schedules, and executions that use our runtime system may diverge from the expected schedule after a data race.

Further, our system produces sets of input-covering schedules that can contain *infeasible schedules*—this happens because our schedule enumeration algorithm is built on the symbolic execution techniques mentioned above. As a result, verification tools that analyze input-covering schedules may report false positives. We observed this effect in practice: as described in Section 3.8.1, our deadlock checker reported infeasible deadlocks for two applications.

Finally, our schedule enumeration algorithm can suffer from combinatorial explosion, making it impractical for some programs. We observed severe combinatorial explosion for one benchmark application, as described in Section 3.8.1.4.

5.3 Future Opportunities

Looking forward, this dissertation opens a number of new research directions that we summarize below:

Exploiting symbolic execution from arbitrary contexts. This dissertation showed one use case for symbolic execution from arbitrary contexts—to aid an algorithm for enumerating input-covering schedules (as described in Section 3.3.3)—but the same techniques are useful in other settings. For example, our techniques can enable *execution reconstruction*, in which the goal is to use very little recorded information (such as a crash dump and little else) to reconstruct the multithreaded path that led to an observed software failure. Other promising applications are described in Section 2.2.

Combining dataflow analyses with symbolic execution. This dissertation demonstrated the benefits of combining dataflow analyses with symbolic execution, specifically towards solving the "symbolic execution from arbitrary contexts" problem. We evaluated two dataflow analyses, but many dataflow analyses have been proposed and there is room to study *which* dataflow analyses are most profitable in this setting. There is also the potential to combine dataflow analyses and symbolic execution in other ways. For example, rather than using dataflow analyses to summarize code at program entry, we might use dataflow analyses to summarize code in the *middle* of a symbolic execution, such as to summarize the effects of a system call or a library call. This potential integration strategy has not yet been explored.

Exploiting input-covering schedules for testing and verification. This dissertation described a deadlock checker that exploited input-covering schedules very simple way. We believe that other, more powerful checkers, testing tools, and verification tools can be built to exploit input-covering schedules. One path forward is suggested by the related work summarized in Section 4.4.

Defining "schedules" for input-covering schedules. Finally, it is unfortunate that our schedule enumeration algorithm suffers from combinatorial explosion in some applications. How can this be avoided? One path forward is to change the definition of *schedules*. For example, rather than defining schedules as happens-before graphs, as we did throughout this dissertation, we might instead define schedules as constraints on pairs of accesses, similarly to systems like PSets [99] (see Section 4.2 for a discussion). Such an alternate definition may lead the way towards a more scalable system.

BIBLIOGRAPHY

[1] μclibc: an Open-Source C Library. `http://www.uclibc.org/`.

[2] NVidia's CUDA Language. `http://developer.nvidia.com/cuda`.

[3] OpenMP 4.0 Application Program Interface. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, 2013.

[4] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-Happen-in-Parallel Analysis of X10 Programs. In *PPoPP*, 2007.

[5] Alexander Aiken and David Gay. Barrier Inference. In *POPL*, 1998.

[6] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Symbolic Execution with Abstract Subsumption Checking. In *SPIN*, 2006.

[7] Joe Armstrong. The Development of Erlang. In *ICFP*, 1997.

[8] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient System-Enforced Deterministic Parallelism. In *OSDI*, 2010.

[9] Rajkishore Barik. Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs. In *Proceedings of the 18th international conference on Languages and Compilers for Parallel Computing (LCPC)*, 2006.

[10] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.

[11] Tom Bergan, Luis Ceze, and Dan Grossman. Input-Covering Schedules for Multithreaded Programs. In *OOPSLA*, 2013.

[12] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The Deterministic Execution Hammer: How Well Does it Actually Pound Nails? In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2011.

[13] Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. Technical Report UW-CSE-13-08-01, Univ. of Washington.

[14] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steve Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.

[15] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe and Efficient Concurrent Programming. In *OOPSLA*, 2009.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[17] H.-J. Boehm and S. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.

[18] Hans-J. Boehm. Simple Garbage-Collector-Safety. In *PLDI*, 1996.

[19] Hans-J. Boehm. Position Paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *RACES*, 2008.

[20] Hans-J. Boehm. How to Miscompile Programs with “Benign” Data Races. In *HotPar*, 2011.

[21] Sascha Böhme and MichałMoskal. Heaps and Data Structures: A Challenge for Automated Provers. In *Proceedings of the 23rd International Conference on Automated Deduction*, 2011.

[22] Michael D. Bond and Kathryn S. McKinley. Probabilistic Calling Context. In *OOPSLA*, 2007.

[23] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS*, 2008.

[24] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *EuroSys*, 2011.

[25] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*, 2010.

[26] R. M. Burstall. Some Techniques for Proving Correctness of Programs which Alter Data Structures. *Machine Intelligence*, 7:23–50, 1972.

[27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[28] Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A Reachability Predicate for Analyzing Low-Level Software. In *TACAS*, 2007.

[29] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Partial Replay of Long-Running Applications. In *FSE*, 2011.

[30] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In *ASPLOS*, 2011.

[31] Katherine E. Coons, Madanlal Musuvathi, and Kathryn S. McKinley. Bounded Partial-Order Reduction. In *OOPSLA*, 2013.

[32] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. In *SOSP*, 2007.

[33] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, 1977.

[34] Heming Cui, Jingyue Wu, Chia che Tsai, and Junfeng Yang. Stable Deterministic Multithreading Through Schedule Memoization. In *OSDI*, 2010.

[35] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient Deterministic Multithreading through Schedule Relaxation. In *SOSP*, 2011.

[36] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *PLDI*, 2002.

[37] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[38] Jospeh Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ASPLOS*, 2011.

[39] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs. In *PLDI*, 2011.

[40] Matthew B. Dwyer and Lori A. Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. In *FSE*, 1994.

[41] Laura Effinger-Dean, Hans-Jeurgen Boehm, Pramod Joisha, and Dhruva Chakrabarti. Extended Sequential Reasoning for Data-Race-Free Programs. In *Workshop on Memory Systems Performance and Correctness (MSPC)*, 2011.

[42] Laura Effingernger-Dean. *Interference-Free Regions and Their Application to Compiler Optimization and Data-Race Detection*. PhD thesis, Computer Science and Engineering, University of Washington, 2012.

[43] Bassem Elkarablieh, Patrice Godefroid, and Michael Y. Levin. Precise Pointer Reasoning for Dynamic Test Generation. In *ISSTA*, 2009.

[44] Peter Ericsson. pfscan: a parallel file scanner. `http://ostatic.com/pfscan`.

[45] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[46] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *CAV*, 2007.

[47] Patrice Godefroid. Compositional Dynamic Test Generation. In *POPL*, 2007.

[48] Patrice Godefroid. Higher-Order Test Generation. In *PLDI*, 2011.

[49] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[50] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium*, 2008.

[51] Patrice Godefroid and Daniel Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *ISSTA*, 2011.

[52] T. Hansen, P. Schachte, and H. Sondergaard. State Joining and Splitting for the Symbolic Execution of Binaries. In *Intl. Conf. on Runtime Verification (RV)*, 2009.

[53] D. Hower, P. Dudnik, D. Wood, and M. Hill. Calvin: Deterministic or Not? Free Will to Choose. In *HPCA*, 2011.

[54] IEEE and The Open Group. IEEE Standard 1003.1-2001. 2001.

[55] ISO. *C Language Standard, ISO/IEC 9899:2011.* 2011.

[56] Tor E. Jeremiassen and Susan J. Eggers. Static Analysis of Barrier Synchronization in Explicitly Parallel Programs. In *PACT*, 1994.

[57] Horatiu Jula, Pinar Tozun, and George Candea. Communix: A Framework for Collaborative Deadlock Immunity. In *DSN*, 2011.

[58] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *OSDI*, 2008.

[59] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *CAV*, 2007.

[60] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *TACAS*, 2003.

[61] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient State Merging in Symbolic Execution. In *PLDI*, 2012.

[62] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *CAV*, 2009.

[63] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *SIGMETRICS*, 2010.

[64] Akash Lal and Thomas Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *CAV*, 2008.

[65] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), July 1978.

[66] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, 28(9), 1979.

[67] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization.* PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005.

[68] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, 2004.

[69] You Li, Zhendong Su, Lingzhang Wang, and Xuandong Li. Steering Symbolic Execution to Less Traveled Paths. In *OOPSLA*, 2013.

[70] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient Deterministic Multithreading. In *SOSP*, 2011.

[71] Yanhonh A. Liu and Scott D. Stoller. From Recursion to Iteration: What are the Optimizations? In *PEPM*, 1999.

[72] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. Efficient Deterministic Multithreading Without Global Barriers. In *PPoPP*, 2014.

[73] Brandon Lucia and Luis Ceze. Cooperative Empirical Failure Avoidance for Multithreaded Programs. In *ASPLOS*, 2013.

[74] Brandon Lucia, Luis Ceze, and Karin Strauss. ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-Variable Atomicity Violations. In *ISCA*, 2010.

[75] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, 2008.

[76] Leonardo De Moura and Nikolaj Bjrner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[77] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[78] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, 2007.

[79] Gleb Naumovich and George S. Avrunin. A Conservative Data Flow Algorithm for Detecting All Pairs of Statements That May Happen in Parallel. In *FSE*, 1998.

[80] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. In *FSE*, 1999.

[81] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[82] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places. In *ASPLOS*, 2009.

[83] Corina S. Pasareanu, Neha Rungta, and Willem Visser. Symbolic Execution with Mixed Concrete-Symbolic Solving. In *ISSTA*, 2011.

[84] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing Procedures in Concurrent Programs. In *POPL*, 2004.

[85] Shaz Qadeer and Dinghao Wu. KISS: Keep It Simple and Sequential. In *PLDI*, 2005.

[86] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs. In *ASPLOS*, 2009.

[87] Zvonimir Rakamarić and Alan J. Hu. A Scalable Memory Model for Low-Level Code. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2009.

[88] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Rahul Nagpal, Karthik Pattabiraman, and Benjamin Zorn. Detecting and Tolerating Asymmetric Races. In *PPoPP*, 2009.

[89] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, 1995.

[90] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a Concolic Unit Testing Engine for C. In *FSE*, 2005.

[91] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, 2002.

[92] Nikolai Tillmann and Jonathan de Halleux. Pex - White Box Test Generation for .NET. In *Tests and Proofs (TAP)*, 2008.

[93] Sam Tobin-Hochstadt and David Van Horn. Higher-Order Symbolic Execution via Contracts. In *OOPSLA*, 2012.

[94] Jan Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *FSE*, 2007.

[95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[96] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *PLDI*, 2012.

[97] M. Xu, M. Hill, and R. Bodik. A Regulated Transitive Reduction for Longer Memory Race Recording. In *ASPLOS*, 2006.

[98] Junfeng Yang, Heming Cui, and Jingyue Wu. Determinism Is Overrated: What Really Makes Multithreaded Programs Hard to Get Right and What Can Be Done About It. In *HotPar*, 2013.

[99] Jie Yu and Satish Narayanasamy. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *ISCA*, 2009.

[100] Jie Yu and Satish Narayanasamy. Tolerating Concurrency Bugs Using Transactions as Lifeguards. In *Micro*, 2010.

[101] Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea. Automated Debugging for Arbitrarily Long Executions. In *HotOS*, 2013.

[102] Yuan Zhang and Evelyn Duesterwald. Barrier Matching for Programs With Textually Unaligned Barriers. In *PPoPP*, 2007.

Appendix A

CONCRETE SEMANTICS OF *SimpThreads*

A.1 Program state in the concrete semantics

$$\begin{array}{rcll}
\mathcal{H} & : & \mathit{Loc} \rightarrow \{\text{fields} : (\mathbb{Z} \rightarrow \mathit{Value})\} & \textit{(heap)} \\
\overline{\mathcal{Y}} & : & \mathit{ThreadId} \rightarrow \text{Stack of } (\mathit{Var} \rightarrow \mathit{Value}) & \textit{(local variables)} \\
\mathit{CallCtx} & : & \mathit{ThreadId} \rightarrow \text{Stack of } \mathit{StmtLabel} & \textit{(calling contexts)} \\
T^{\mathit{Curr}} & : & \mathit{ThreadId} & \textit{(current thread)} \\
T^{E} & : & \text{Set of } \mathit{ThreadId} & \textit{(enabled threads)} \\
\mathit{WQ} & : & \text{List of } (\mathit{Value}, \mathit{ThreadId}) & \textit{(global wait queue)} \\
\mathtt{L}^{+} & : & \mathit{ThreadId} \rightarrow \text{Set of } \mathit{Value} & \textit{(acquired locksets)}
\end{array}$$

A.2 Auxiliary Functions

$\mathit{eval} : ((\mathit{Var} \rightarrow \mathit{Value}) \times \mathit{Expr}) \rightarrow \mathit{Value}$

This is just as in the symbolic semantics (Section 2.3.1), except that all expressions can be reduced to values as symbolic constants do not appear in the concrete language.

$\mathit{wqGetOne} : (\mathit{WaitQueue}, \mathit{Value}) \rightarrow (\mathit{ThreadId}, \mathit{WaitQueue})$

Given $\mathit{wqGetOne}(\mathit{WQ}, v)$, we walk the global queue WQ and return the first thread (t) waiting on the queue at address v. If found, we return a pair (t, WQ'), where WQ' is WQ with t dequeued. If the queue at address v is empty, we return ϵ.

$\mathit{wqGetAll} : (\mathit{WaitQueue}, \mathit{Value}) \rightarrow (\text{Set of } \mathit{ThreadId}, \mathit{WaitQueue})$

As above, except we dequeue all threads waiting on the queue at address v. If that queue is empty, we return (ϵ, WQ).

$\mathit{wqAppend} : (\mathit{WaitQueue}, \mathit{Value}, \mathit{ThreadId}) \rightarrow \mathit{WaitQueue}$

Given $\mathit{wqGetOne}(\mathit{WQ}, v, t)$, we append the pair (v, t) to WQ.

A.3 Statement evaluation rules

The $\Longrightarrow$ relation, defined below, is equivalent to *step* in the symbolic semantics except that $\Longrightarrow$ never forks the concrete state. We do not include semantics for the annotations `barrierInit` and `barrierArrive`, as these are no-ops during symbolic execution. We include a few shorthands for brevity: We elide a domain from a rule if the domain is not used or updated by the rule. We use $\mathcal{Y}$ (without the overline) to refer to the current stack frame (namely, the youngest stack frame of $\overline{\mathcal{Y}}(T^{Curr})$). We write $M[k \mapsto v]$ to assign $k = v$ in map M, and we write M/k to remove the pair $k \mapsto M(k)$ from map M. We also elide rules that check for memory errors such as out-of-bounds accesses and data races, as the details of these checkers are orthogonal to this dissertation.

$$\boxed{\mathcal{H}; \overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr}; T^{E}; \mathit{WQ}; \mathtt{L}^{+}; \mathtt{B}^{\mathtt{cnts}} | \mathit{Stmt} \Longrightarrow \mathcal{H}'; \overline{\mathcal{Y}}'; \mathit{CallCtx}'; T^{Curr}_{new}; T^{E}_{new}; \mathit{WQ}'; \mathtt{L}^{+}_{new}; \mathtt{B}^{\mathtt{cnts}}_{new}}$$

$$\frac{\begin{array}{c} \mathit{size}(\mathit{CallCtx}(T^{Curr})) > 1 \qquad \overline{\mathcal{Y}}' = \overline{\mathcal{Y}}[T^{Curr} \mapsto \mathit{pop}(\overline{\mathcal{Y}}(T^{Curr}))] \\ \mathit{CallCtx}' = \mathit{CallCtx}[T^{Curr} \mapsto \mathit{pop}(\mathit{CallCtx}(T^{Curr}))] \qquad \gamma' = \mathit{top}(\mathit{CallCtx}'(T^{Curr})) \\ \mathit{Stmts}(\gamma') = r \leftarrow e_f(e_a^*) \qquad \mathit{eval}(\mathcal{Y}, e) = v \qquad \mathcal{Y}' = \mathit{top}(\overline{\mathcal{Y}}'(T^{Curr}))[r \mapsto v] \end{array}}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr}; T^{E} | \mathtt{return}\ e \Longrightarrow \overline{\mathcal{Y}}'[T^{Curr} \mapsto \mathit{replaceTop}(\overline{\mathcal{Y}}'(T^{Curr}), \mathcal{Y}')]; \mathit{CallCtx}'; T^{Curr}; T^{E}}$$

$$\frac{\mathit{size}(\mathit{CallCtx}(T^{Curr})) = 1 \qquad T^{Curr}_{new} \in T^{E} \qquad T^{Curr}_{new} \neq T^{Curr}}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr}; T^{E}; \mathtt{L}^{+} | \mathtt{return}\ e \Longrightarrow \overline{\mathcal{Y}}[T^{Curr} \to \epsilon]; \mathit{CallCtx}[T^{Curr} \to \epsilon]; T^{Curr}_{new}; T^{E}/T^{Curr}; \mathtt{L}^{+}[T^{Curr} \to \epsilon]}$$

$$\frac{\mathit{size}(\mathit{CallCtx}(T^{Curr})) = 1 \qquad T^{E} = \{T^{Curr}\}}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr}; T^{E}; \mathtt{L}^{+} | \mathtt{return}\ e \Longrightarrow \epsilon; \epsilon; \epsilon; \epsilon; \epsilon}$$

$$\frac{\mathit{eval}(\mathcal{Y}, e) = i \qquad \gamma = ((i \neq 0)\ ?\ \gamma_t\ :\ \gamma_f)}{\overline{\mathcal{Y}}; \mathit{CallCtx}; T^{Curr} | \mathtt{br}\ e,\ \gamma_t,\ \gamma_f \Longrightarrow \overline{\mathcal{Y}}; \mathit{goto}(\mathit{CallCtx}, T^{Curr}, \gamma); T^{Curr}}$$

$$\frac{\begin{array}{c} eval(\mathcal{Y}, e_f) = f \qquad eval(\mathcal{Y}, e_i) = v_i \\ Funcs(f) = \texttt{func}\ \ f(r_i*)\{\gamma_0 : \ldots\} \qquad \overline{\mathcal{Y}}' = \overline{\mathcal{Y}}[T^{Curr} \mapsto push(\overline{\mathcal{Y}}(T^{Curr}), \{r_i \rightarrow v_i\})] \\ CallCtx' = CallCtx[T^{Curr} \mapsto push(CallCtx(T^{Curr}), \gamma_0)];\ T^{Curr} \end{array}}{\overline{\mathcal{Y}};\ CallCtx;\ T^{Curr} | r \leftarrow e_f(e_i^*) \Longrightarrow \overline{\mathcal{Y}}';\ CallCtx';\ T^{Curr}}$$

$$\frac{\begin{array}{c} eval(\mathcal{Y}, p) = \texttt{ptr}(l, i) \\ (l, \{\text{fields}\}) \in \mathcal{H} \end{array}}{\mathcal{H}; \mathcal{Y} | r \leftarrow \texttt{load}\ \ p \Longrightarrow \mathcal{H}; \mathcal{Y}[r \mapsto \text{fields}(i)]} \qquad \frac{\begin{array}{c} eval(\mathcal{Y}, p) = \texttt{ptr}(l, i) \\ eval(\mathcal{Y}, e) = e' \qquad (l, \{\text{fields}\}) \in \mathcal{H} \end{array}}{\mathcal{H}; \mathcal{Y} | \texttt{store}\ \ p,\ \ e \Longrightarrow \mathcal{H}[l \mapsto \{\text{fields}[i \mapsto e']\}]; \mathcal{Y}}$$

$$\frac{l = fresh\ loc}{\mathcal{H}; \mathcal{Y} | r \leftarrow \texttt{malloc}(e) \Longrightarrow \mathcal{H}[l \mapsto \{\lambda i.\texttt{undef}\}]; \mathcal{Y}[r \mapsto \texttt{ptr}(l, 0)]} \qquad \frac{true}{\mathcal{H}; \mathcal{Y} | \texttt{free}(p) \Longrightarrow \mathcal{H}; \mathcal{Y}}$$

$$\frac{\begin{array}{c} eval(\mathcal{Y}, e_f) = f \\ eval(\mathcal{Y}, e_{arg}) = v \qquad Funcs(f) = \texttt{func}\ \ f(r)\{\gamma_0 : \ldots\} \qquad T_{new} = fresh\ id \end{array}}{\overline{\mathcal{Y}};\ CallCtx;\ T^E | \texttt{threadCreate}(e_f, e_{arg}) \Longrightarrow \overline{\mathcal{Y}}[T_{new} \mapsto \{\{r \rightarrow v\}\}];\ CallCtx[T_{new} \mapsto \gamma_0];\ T^E \cup \{T_{new}\}}$$

$$\frac{T^{Curr}_{new} \in T^E}{T^{Curr};\ T^E | \texttt{yield}() \Longrightarrow\ T^{Curr}_{new};\ T^E}$$

$$\frac{eval(\mathcal{Y}, p) = v \qquad wqAppend(WQ, v, T^{Curr}) = WQ' \qquad T^E = \{T^{Curr}\}}{\overline{\mathcal{Y}};\ T^{Curr};\ T^E;\ WQ | \texttt{wait}(p) \Longrightarrow \overline{\mathcal{Y}}; \epsilon; \{\};\ WQ'}$$

$$\frac{eval(\mathcal{Y}, p) = v \qquad wqAppend(WQ, v, T^{Curr}) = WQ' \qquad T^{Curr}_{new} \in T^E \qquad T^{Curr}_{new} \neq T^{Curr}}{\overline{\mathcal{Y}};\ T^{Curr};\ T^E;\ WQ | \texttt{wait}(p) \Longrightarrow \overline{\mathcal{Y}};\ T^{Curr}_{new};\ T^E / \{T^{Curr}\};\ WQ'}$$

$$\frac{eval(\mathcal{Y}, p) = v \qquad wqGetOne(WQ, v) = \epsilon}{\overline{\mathcal{Y}};\ T^E;\ WQ | \texttt{notifyOne}(p) \Longrightarrow \overline{\mathcal{Y}};\ T^E;\ WQ}$$

$$\frac{eval(\mathcal{Y}, p) = v \qquad wqGetOne(WQ, v) = (hd,\ WQ')}{\overline{\mathcal{Y}};\ T^E;\ WQ | \texttt{notifyOne}(p) \Longrightarrow \overline{\mathcal{Y}};\ T^E \cup \{hd\};\ WQ'}$$

$$\frac{eval(\mathcal{Y}, p) = v \qquad wqGetAll(WQ, v) = \{T_{woke}, WQ'\}}{\overline{\mathcal{Y}};\, T^{E};\, WQ|\texttt{notifyAll}(p) \Longrightarrow \overline{\mathcal{Y}};\, T^{E} \cup \{T_{woke}\};\, WQ'}$$

$$\frac{eval(\mathcal{Y}, p) = v}{\overline{\mathcal{Y}};\, T^{Curr}|\texttt{acquire}(p) \Longrightarrow \overline{\mathcal{Y}};\, T^{Curr}; \texttt{L}^{+}[T^{Curr} \mapsto \texttt{L}^{+}(T^{Curr}) \cup \{v\}]}$$

$$\frac{eval(\mathcal{Y}, p) = v}{\overline{\mathcal{Y}};\, T^{Curr}; \texttt{L}^{+}|\texttt{release}(p) \Longrightarrow \overline{\mathcal{Y}};\, T^{Curr}; \texttt{L}^{+}[T^{Curr} \mapsto \texttt{L}^{+}(T^{Curr})/\{v\}]}$$

Appendix B

SOUNDNESS AND COMPLETENESS OF THE SYMBOLIC SEMANTICS

We restate and expand on Definition 1 and then restate and give a proof for Theorem 1. These were first stated in Section 2.7.

Definition 1 (Correspondence of concrete and symbolic states)**.** *We say that symbolic state S_S* models *concrete state S_K* under *constraint C if there exists an assignment Σ that assigns all symbolic constants in S_S to values such that (a) Σ is a valid assignment under the constraint C, and (b) the application of Σ to S_S produces a state S'_S that is* partially-equivalent *to S_K (as defined below).*

The above definition relies on a notion of *partial* equivalence between S'_S and S_K, rather than true equivalence, because we expand the symbolic memory graph lazily (recall Section 2.4.1). Thus, the symbolic heap may contain a subset of the objects contained in the concrete heap. Our notion of partial equivalence considers only this overlapping subset of S'_S and S_K. Hence, to determine whether S'_S and S_K are partially equivalent, we must construct a mapping, λ, that maps locations l_S in S'_S to isomorphic locations l_K in S_K. This is actually a many-to-one mapping as multiple locations in the symbolic heap can alias a single location in the concrete heap, due to our representation of aliasing in $S_S.\mathcal{H}$ (again, recall Section 2.4.1).

We give a complete definition of partial equivalence below:

Definition 2 (Partial equivalence of states)**.** *Given a symbolic state S_S, an assignment Σ, and a concrete state S_K, let S'_S be the state produced when Σ is applied to S_S. We assume, without loss of generality, that the set of locations used by values in S'_S is disjoint from the set of locations used by values in S_K.*

We check if the memory graph in S'_S is isomorphic to a subset of the memory graph *in S_K, where each memory graph is defined relative to the pointer roots in $\overline{\mathcal{Y}}$. If no such isomorphism exists, then S'_S and S_K are not partially equivalent.*

Specifically, we must find a correspondence $\lambda(l_S) = l_K$ between heap locations l_S from S'_S and l_K from S_K such that λ satisfies the following conditions. We say that S'_S and S_K are partially equivalent *if and only if such a λ exits.*

- *$\forall l_S \in S'_S$, $l_S \in \lambda$. That is, if location l_S is used by any value in S'_S (not just in $S'_S.\mathcal{H}$), then it must have a mapping in λ.*

- *$\forall l_K \in S_K.\mathcal{H}$, if there does not exist an l_S such that $\lambda(l_S) = l_K$ and $l_S \in S'_S.\mathcal{H}$, then it should be possible to "expand" some symbolic pointer in S'_S to reach such an l_S. Specifically, there should exist an x such that $\Sigma(x) = \boldsymbol{ptr}(l'_S, i)$ and $\lambda(l'_S) = l'_K$ (but $l'_S \notin S'_S.\mathcal{H}$, as x should be unexpanded), where either (a) $l'_K = l_K$, or (b) $l'_K \neq l_K$, but l_K is reachable from l'_K in $S_K.\mathcal{H}$. In case (a), we expand x directly to a heap object l_S (where $\lambda(l_S) = l_K$), and in case (b), we expand x to some heap object l'_S from which object l_S is transitively reachable.*

- *$\forall l_S \in S'_S.\mathcal{H}$, the heap object at $S'_S.\mathcal{H}(l_S)$ matches the heap object at $S_K.\mathcal{H}(\lambda(l_S))$. We say that two values v_S and v_K "match" if either $v_S = v_K$ or if $v_S = \boldsymbol{ptr}(l_S, i_S)$ and $v_K = \boldsymbol{ptr}(l_K, i_K)$ where $\lambda(l_S) = l_K$ and $i_S = i_K$.*

- *$T \in S'_S.\overline{\mathcal{Y}}$ if and only if $T \in S_K.\overline{\mathcal{Y}}$, and further, $\forall T \in S'_S.\overline{\mathcal{Y}}$, the stack $S'_S.\overline{\mathcal{Y}}(T)$ matches the youngest stack frames in $S_K.\overline{\mathcal{Y}}(T)$. This definition allows S_K to have deeper stack frames than S'_S, as the call stacks in S'_S may be underspecified (recall Section 2.3.2).*

- *$T \in S'_S.CallCtx$ if and only if $T \in S_K.CallCtx$, and further, $\forall T \in S'_S.CallCtx$, the stack $S'_S.CallCtx(T)$ matches the youngest stack frames in $S_K.CallCtx(T)$. As above, this definition allows S_K to have deeper stack frames than S'_S.*

- *T^{Curr} and T^{E} match exactly in S'_S and S_K.*

- *WQ matches exactly in S'_S and S_K. If the symbolic WQ uses initial waiter timestamps $\{x_0, x_1, \cdots\}$ (recall Section 2.5.2), then those initial entries of $S'_S.WQ$ are ordered by the concrete values of their respective timestamps as assigned by Σ.*

- *$T \in S'_S.\mathtt{L}^+$ if and only if $T \in S_K.\mathtt{L}^+$, and further, $\forall T \in S'_S.\mathtt{L}^+$, $S_K.\mathtt{L}^+(T) \subseteq S'_S.\mathtt{L}^+(T)$. This definition allows $S'_S.\mathtt{L}^+$ to be a conservative over-approximation of $S_K.\mathtt{L}^+$.*

Theorem 1 (Soundness and completeness of symbolic execution)**.** *Consider an initial program context, an initial concrete state S_K for that context, and an initial symbolic state S_S:*

– ***Soundness:*** *If symbolic execution from S_S outputs a pair (p, C), then for all S_K such that S_S models S_K under C, concrete execution from S_K must follow path p as long as context switches happen exactly as specified by path p.*

– ***Completeness:*** *If concrete execution from S_K follows path p, then for all S_S such that S_S models S_K under $S_S.C$, symbolic execution from S_S will either (a) output a pair (p, C), for some C, or (b) encounter a query that the SMT solver cannot solve.*

Proof. First, we state our assumptions. We assume the underlying points-to analysis is sound. We assume that *isSat* is sound, though not necessarily complete. We assume that synchronization libraries are correctly implemented, as otherwise, the invariants described in Section 2.5.3 would be incorrect. Note that the initial symbolic state S_S is a given in the statement of the theorem, thus our theorem implicitly assumes correctness of the dataflow analyses used to construct that initial symbolic state. Finally, our completeness proof is valid only for paths p in which no thread continues executing after returning from its initial stack frame (recall from Section 2.3.2 that symbolic execution does not continue beyond this point). We now prove the theorem:

Soundness. The proof proceeds by induction over the length of an execution trace, with symbolic and concrete executions proceeding in lockstep. The base case is given. In the inductive cases, symbolic execution from S_S and concrete execution from S_K take the next action on path p, resulting in states S'_S and S'_K, respectively, and we show that the inductive hypothesis is maintained (*i.e.*, that S'_S models S'_K under C).

Importantly, our proof must demonstrate that, whenever the symbolic execution makes a choice (such as at branch statements), the path constraint $S'_S.C$ and execution trace $S'_S.\mathit{path}$ must be updated in such a way that the concrete execution is *forced* to make the same choice. That is, the concrete execution cannot make a different choice unless it is *not* true that S_S models S_K under C.

In this proof, as a shorthand, we say that an expression in the symbolic state is *consistent* with a value in the concrete state if there exists a valid assignment Σ such that, if Σ is applied to the symbolic expression, then the resulting value matches the concrete value. (This is merely an application of Definition 1.)

We give one case for each possible program statement:

- **return stmt** *"return e"*: The symbolic and concrete executions pop call stacks in the same mechanical way, hence their updates will be consistent. The consistency of return value e in the symbolic and concrete executions follows from the inductive hypothesis. If the current thread T is returning from its final stack frame in $S_S.\mathit{CallCtx}$, then there are three special cases: (1) T is the only thread in T^E, in which case symbolic execution halts and the path completes; otherwise (2) $|S_K.\mathit{CallCtx}| = 1$ and T exits from both the symbolic and concrete executions; or (3) $|S_K.\mathit{CallCtx}| > 1$, in which case the call stacks are underspecified and symbolic execution will not schedule T again (recall Section 2.3.2). In case (3), on a technical point, to maintain the inductive hypothesis, we must ensure that $T \in S_S.\overline{\mathcal{Y}}$ if and only if $T \in S_K.\overline{\mathcal{Y}}$ (recall Definition 2)—we do this by not removing T from $\overline{\mathcal{Y}}$ or $\mathit{CallCtx}$ when T exits (*e.g.*, see the second rule for `return` in the concrete semantics).

- **call stmt** *"$r \leftarrow e_f(e^*)$"*: The symbolic and concrete executions push call stacks in the same mechanical way, hence their updates will be consistent. The consistency of values e^* in the symbolic and concrete executions follows from the inductive hypothesis. At indirect calls, if the symbolic execution invokes function f, it updates the path constraint to $C' = (C \wedge e_f = f)$. By the inductive hypothesis, the concrete execution must derefence a function pointer e_f that is consistent with C' in the symbolic execution. Thus, the concrete execution must invoke the same function.

- **branch stmt** *"br* e, γ_t, γ_f*"***:** When symbolic execution takes the branch to γ_t, it updates the path constraint to $C' = C \wedge e$ (similarly, to $C' = C \wedge \neg e$ for γ_f). This constraint is sufficiently narrow to force the concrete execution to take the same branch.

- **the first access of symbolic pointer** x**:** Suppose the symbolic execution is about to access pointer x for the first time. In this case, there does not yet exist a record $\{x, l_x, n_x\} \in S_S.\mathcal{A}$. Such a record will be appended and a new primary object l_x will be allocated. We must show that this update is performed in such a way that the resulting symbolic state will be consistent with the concrete state. Specifically, consider all pairs (l_S, i) such that $\Sigma(x) = \mathtt{ptr}(l_S, i)$, where Σ is a valid assignment as in Definition 1. We first show that, for all such pairs (l_S, i), $x = \mathtt{ptr}(l_S, i) \implies S_S.\mathcal{H}(l_S).\mathit{fields} = S_S.\mathcal{H}(l_x).\mathit{fields}$. This is ensured by Equation (2.1), which ensures that l_x has an initial state that matches all possible aliases. (Note that this argument implicitly assumes that the set of aliases is soundly chosen—this follows from our assumption that our underlying static points-to analysis is sound.) We next show that, for any l_K in the concrete heap that corresponds to one possible l_S, the objects at l_K and l_S are consistent. This follows directly from the inductive hypothesis when $l_S \neq l_x$, and otherwise, it follows from the fact that we assign each new symbolic heap object a fresh symbolic array (recall Equation (2.1)).

- **memory access stmt** *"$r \leftarrow$load(p) or store(p, e)"***:** First we assume that the access does not have a memory error. By the inductive hypothesis, the pointer p and value e are consistent in the symbolic and concrete executions. We consider three cases:

  (1) p has the form $\mathtt{ptr}(l, e_{\mathit{off}})$ in the symbolic state. Note that this case can arise only if l was allocated by a call to `malloc` during symbolic execution. That is, l cannot alias any symbolic pointer x in S_S, and we do not need to consider the correctness of

aliases. Thus, for this case, loads and stores perform the same mechanical action in both semantics, and we conclude that the resulting states are consistent.

(2) p has the form x or $\texttt{ptradd}(x, e_{\mathit{off}})$ in the symbolic state and the action is a load. Suppose that $\Sigma(x) = \texttt{ptr}(l_S, i)$, where Σ satisfies Definition 1. We must show that the symbolic and concrete executions read consistent values. By the inductive hypothesis combined with the above case for the "first access of x," location l_S in the symbolic heap must be consistent with some corresponding location l_K in the concrete heap. Thus, we conclude that the concrete and symbolic loads will return consistent values.

(3) p has the form x or $\texttt{ptradd}(x, e_{\mathit{off}})$ in the symbolic state and the action is a store. Suppose again that $\Sigma(x) = \texttt{ptr}(l_S, i)$. By the inductive hypothesis combined with the above case for the "first access of x," location l_S in the symbolic heap must be consistent with some corresponding location l_K in the concrete heap. We must show that l_S and l_K are updated in a consistent manner. This follows from, first, the fact that symbolic execution updates all $l \in S_S.\mathit{lookupAliases}(x)$, and second, from the assumption that our static points-to analysis is sound, which implies that the set of aliases in $S_S.\mathcal{A}$ soundly covers all possible aliases in the symbolic heap. Thus, we conclude that the concrete and symbolic heaps are updated in a consistent manner.

As we do not give detailed semantics for memory errors in this disseration, our proof will not discuss memory errors in detail. Briefly, at each access of p, the symbolic execution forks into two states—one with a memory error and one without—and updates the path constraint C in each state to describe each case. For soundness, the constraints must be narrow enough so that, if the symbolic execution does (or does not) follow a path with a memory error, then the concrete execution must (or must not) follow a path with the same memory error.

- **allocator stmt** *"malloc(e) or free(p)"*: Again, as we do not discuss memory errors in detail in this dissertation, these cases are trivial as the concrete and symbolic executions both perform the same mechanical action.

- **threadCreate stmt:** The consistency of values e_f and e_{arg} in the symbolic and concrete executions follows from the inductive hypothesis. Hence, as this statement has the same mechanical action in both semantics, the resulting states are consistent.

- **yield stmt:** By the inductive hypothesis, $S_S.T^E = S_K.T^E$. Hence, whichever next T^{Curr} is selected in the symbolic execution can also be selected by the concrete execution. Further, we conclude that the *same* next T^{Curr} will be selected by the concrete execution, as our inductive hypothesis assumes that context switches in the concrete execution are dictated precisely by the *path* output by the symbolic execution.

- **wait stmt** *"wait(p)"***:** The consistency of value p in the symbolic and concrete executions follows from the inductive hypothesis. Hence, WQ is updated the same way in both executions. Further, as with *yield*, the next T^{Curr} selected by the symbolic execution will also be selected by the concrete execution.

- **notify stmt** *"notifyOne(p) or notifyAll(p)"***:** By the inductive hypothesis, the wait queues WQ are consistent in both the symbolic and concrete states. Suppose the symbolic execution wakes a set of threads T^{woke} (which includes at most one thread for `notifyOne`). We submit that the constraints described in Section 2.5.2 are sufficiently narrow to force the concrete execution to wake the exact same set of threads, T^{woke}.

- **annotation** *"acquire(p) or release(p)"***:** By the inductive hypothesis, $\mathtt{L}^+$ is consistent in both the symbolic and concrete states. On `acquire`, $\mathtt{L}^+$ is updated in mechanically the same way in both the symbolic and concrete semantics (recall Section 2.5.3). On `release`, the symbolic semantics removes p from $\mathtt{L}^+(T^{Curr})$ only if there exists a lock in $\mathtt{L}^+(T^{Curr})$ that must-equal p given the current path constraint. This makes the symbolic $\mathtt{L}^+$ an over-approximation of the concrete $\mathtt{L}^+$, which is allowed by Definition 2.

Completeness. The theorem trivially holds when *isSat* encounters an unsolvable query. Thus, in the remainder of this proof, we assume that *isSat* will soundly resolve any query it is given.

As above, the proof proceeds by induction over the length of an execution trace, with symbolic and concrete executions proceeding in lockstep. The base case is given. In the inductive cases, symbolic execution from S_S and concrete execution from S_K take the next action on path p, resulting in states S'_S and S'_K, respectively, and we show that the inductive hypothesis is maintained (*i.e.*, that S'_S models S'_K under $S'_S.C$). While the soundness proof required constraints to be sufficiently *narrow* to force concrete execution down a specific path, here we require constraints to be sufficiently *wide* so that the symbolic execution can cover all paths that might be followed during the concrete execution.

We again have one case for each possible program statement:

- **call stmt** *"$r \leftarrow e_f(e^*)$"***:** As before, we argue that the symbolic and concrete executions push call stacks in the same mechanical way, hence their updates will be consistent. At indirect calls, if the concrete execution invokes function f, then the symbolic execution must cover a path that invokes function f. This follows, first, from the inductive hypothesis (the expression e_f in the symbolic state is consistent with f), and second, from the assumed soundness of our static points-to analysis that selects the set of possible target functions for this call site.

- **branch stmt** *"br e, γ_t, γ_f"***:** When symbolic execution takes the branch to γ_t, it updates the path constraint to $C' = C \wedge e$ (similarly, to $C' = C \wedge \neg e$ for γ_f). These two constraints are sufficiently wide to cover all possible paths that the concrete execution may take.

- **memory access stmt or allocator stmt:** In the absence of memory errors, we can reuse the same argument from the soundness proof to argue that the resulting states are consistent. In the case of memory errors, recall again that at each access of p, the symbolic execution forks into two states—one with a memory error (S'_S) and one without (S''_S). For completeness, if the concrete execution encounters a memory error, then the updated path constraints in $S'_S.C$ must be wide enough so that, as execution proceeds from S'_S, the symbolic execution will encounter the same memory error.

- **yield stmt:** By the inductive hypothesis, $S_S.T^E = S_K.T^E$. Hence, whichever next T^{Curr} is selected in the concrete execution can also be selected by the symbolic execution.

- **notify stmt:** By the inductive hypothesis, the wait queues WQ are consistent in both the symbolic and concrete states. Suppose the concrete execution wakes a set of threads T^{woke} (which includes at most one thread for `notifyOne`). We submit that the constraints described in Section 2.5.2 are sufficiently wide so that, in at least one forked state, the symbolic execution will explore a path in which the exact same set of threads, T^{woke}, is woken.

- **return stmt, threadCreate stmt, wait stmt, or annotation:** In these cases, the concrete and symbolic semantics perform essentially the same mechanical updates. Hence, similarly to the proof cases stated above under *soundness*, the theorem holds for these statements. (Further, for `wait`(p), we observe that, as for `yield`(), any next T^{Curr} chosen by the concrete execution can also be chosen in the symbolic execution.)

$\square$