© 2006 by Koushik Sen. All rights reserved.

SCALABLE AUTOMATED METHODS FOR DYNAMIC PROGRAM ANALYSIS

BY

KOUSHIK SEN

B.Tech., Indian Institute of Technology at Kanpur, 1999
M.S., University of Illinois at Urbana Champaign, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

Testing using manually generated test cases is the primary technique used in industry to improve reliability of software—in fact, such *ad hoc* testing accounts for over half of the typical cost of software development. We propose new methods for systematically and automatically testing sequential and concurrent programs. The methods are based on three new techniques: *concolic testing*, *race-detection and flipping*, and *predictive monitoring*. Concolic testing combines concrete and symbolic testing to avoid redundant test cases as well as false warnings. Concolic testing can catch generic errors such as assertion violations, uncaught exceptions, and segmentation faults.

Large real-world programs are almost always concurrent. Because of the inherent non-determinism of such programs, testing is notoriously hard. We extend concolic testing with a method called *race-detection and flipping*, which provides ways of reducing, often exponentially, the exploration space for concolic testing. This combined method provides the first technique to effectively test concurrent programs with complex data inputs.

Concolic testing may also be combined with formal specifications by using *runtime monitors*. Runtime monitors are small software units which are synthesized automatically from the formal specification for the software and weaved into the code to dynamically check if the specification is violated. For multi-threaded concurrent programs, we developed a novel technique which allows efficient *predictive monitoring* to enable the detection of a violation by observing some related, but possibly bug-free execution of a concurrent program. Predictive monitoring dramatically improves the efficiency of testing.

Based on the above methods we have developed tools for testing both C and Java programs. We have used the tools to find bugs in several real-world software systems including SGLIB, a popular C data structure library used in a commercial tool, implementations of the Needham-Schroeder protocol and the TMN protocol, the scheduler of Honeywell's DEOS real-time operating system, and the Sun Microsystems' JDK 1.4 collection framework.

To my parents and my wife.

Acknowledgment

First I would like to thank my advisor Gul Agha. His exemplary guidance, his far reaching vision, and his depth and breadth of knowledge has facilitated me to become a researcher. He gave me full freedom and unconditional support to explore and develop diverse ideas throughout my graduate education. The best way for me to express my gratitude towards him is to try to become what he has been to me: teacher, mentor, guide, collaborator, and friend.

I thank José Meseguer for his constant guidance and feedback in my research. My research has benefited considerably from stimulating discussions with him and from our many years of collaboration on probabilistic rewriting theories and statistical methods. His classes on automated software verification introduced me to the whole area of software reliability.

I thank Grigore Roşu for his strong encouragement and guidance in my research. He introduced me with the notion of scalable and light weight formal methods for improving software quality. This notion over time shaped the primary theme of my research. His expertise on runtime verification and scalable formal methods influenced my research on combining ad hoc testing and rigorous formal methods. His strong collaboration helped me to develop the whole area of predictive runtime monitoring and decentralized monitoring of distributed systems.

I would like to thank Mahesh Viswanathan for his constant mentoring and guidance over the last five years. He introduced me to the whole world of model checking and probabilistic methods, and has always provided me with key insights and whole-hearted support on my research and career goals. My research on statistical and probabilistic model checking considerably benefited from his collaboration.

I owe special thanks to Klaus Havelund for mentoring and guiding me from the early stage of my research career. He introduced me to real-world software problems and strongly motivated me to start my research career in software reliability. His collaboration considerably benefited my

research on runtime monitoring.

I am grateful to Martin Rinard for serving on my dissertation committee, and for his valuable comments and questions. I would like to thank Allen Goldberg and Howard Barringer for their strong collaboration in my research on runtime verification.

I am grateful to Patrice Godefroid, Nils Klarlund, and Darko Marinov for their collaboration on the concolic testing work. I would especially like to mention, Nirman Kumar, Prasanna Thati, and Abhay Vardhan, with whom I collaborated on many research projects. Special thanks to Carl Gunter, Michael Greenwald, and Sanjeev Khanna for their collaboration on the verification of probabilistic security protocols project.

I would like to thank everyone with whom I had interactions—both research and otherwise. Specially I would like to thank Nadeem Jamali, Sandeep Uttamchandani, Reza Ziaei, Po-Hao Chang, Myeong-Wuk Jang, YoungMin Kwon, Rajesh Kumar, Timo Latvala, Predrag Tosic, Tom Brown, Liping Chen, Soham Mazumdar, Sameer Sundresh, Kirill Mechitov, WooYoung Kim, Can Zheng, Marcelo Bezerra D'Amorim, Sudarshan M. Srinivasan, Rupak Majumdar, Ranjit Jhala, Sumit Gulwani, Sarfraz Khurshid, Chandrasekhar Boyapati, Tao Xie, Saurabh Sinha, Indranil Gupta, ... I thank the helpful staff at the Department of Computer Science including Erna Amerman, Barb Cicone, Shirley Finke, Mary Beth Kelley, and Dana Kennedy and especially Andrea Whitesell.

Finally and foremost, I would like to thank my wife, Aditi, for her unwavering love, support, and understanding. I thank my parents, whose constant love and encouragement has helped me to be optimistic even in the most difficult days of my life.

Table of Contents

Chapter 1 Introduction 1
1.1 Background 4
1.2 Outline 6

Chapter 2 Overview 7
2.1 Concolic Testing through an Example 7
2.2 The Race-Detection and Flipping Algorithm through an Example 10
2.3 Predictive Monitoring through an Example 12

Chapter 3 Programming and Execution Model 15
3.1 Programming Model 15
3.2 Execution Model 19
3.3 Results in Terms of the Execution Model 31

Chapter 4 Testing Sequential Programs 32
4.1 Concolic Testing 33
4.1.1 Logical Input Map 34
4.1.2 Instrumentation 35
4.1.3 Concolic Execution 37
4.1.4 Bounded Depth-First Search 39
4.1.5 Computing an Input 43
4.1.6 Approximations for Scalable Symbolic Execution 46
4.2 Data Structure Testing 47
4.2.1 Generating Inputs with Call Sequences 47
4.2.2 Solving Data Structure Invariants 48
4.3 Discussion 49
4.3.1 Pointer Casting and Arithmetic 49
4.3.2 Library Functions with Side-Effects 50
4.3.3 Approximating Symbolic Values by Concrete Values 50
4.3.4 Black-Box Library Functions 51
4.3.5 Lazy Initialization 51
4.3.6 Random Initialization 52

Chapter 5 Testing Concurrent Programs 54
5.1 The Race-Detection and Flipping Algorithm 55
5.2 Extending Concolic Testing to Test Concurrent Programs 61
5.2.1 Instrumentation 62

5.2.2 Controlling the Execution of Threads . . . 64
5.2.3 Computing a Schedule and an Input . . . 68
5.3 Extending Concolic Testing with the Race-detection and Flipping Algorithm . . . 68
5.4 A Further Optimization . . . 72
5.5 Discussion . . . 73

Chapter 6 Predictive Monitoring of Concurrent Programs . . . 74
6.1 Monitors for Safety Properties . . . 76
6.2 Relevant Causality . . . 79
6.2.1 Vector Clock Algorithm for Relevant Causality . . . 80
6.3 Runtime Model Generation and Predictive Monitoring . . . 81
6.3.1 Multi-Threaded Computation Lattice . . . 81
6.3.2 Level by Level Analysis of the Computation Lattice . . . 87
6.3.3 Causality Cone Heuristic . . . 91

Chapter 7 Implementation and Case Studies . . . 94
7.1 Implementation . . . 94
7.1.1 Program Instrumentation . . . 94
7.1.2 Utility Functions . . . 96
7.2 Experimental Evaluation . . . 97
7.2.1 Data Structures of CUTE . . . 97
7.2.2 SGLIB Library . . . 98
7.2.3 Java 1.4 Collection Library . . . 100
7.2.4 NASA's Java Pathfinder's Case Studies . . . 105
7.2.5 Needham-Schroeder Protocol . . . 105
7.2.6 TMN Protocol . . . 106

Chapter 8 Related Work . . . 107
8.1 Testing Sequential Programs . . . 107
8.2 Testing Concurrent Programs . . . 108
8.3 Runtime Verification . . . 110

Chapter 9 Conclusion . . . 112
9.1 Summary . . . 112
9.2 Discussion . . . 115
9.2.1 Scalability . . . 115
9.2.2 Program Verification . . . 118

References . . . 120

Author's Biography . . . 129

Chapter 1

Introduction

Software pervades every aspect of our life: businesses, financial services, medical services, communication systems, entertainment, and education are invariably dependent on software. With this increasing dependency on software, we expect software to be reliable, robust, safe, and secure. Unfortunately, at the present time the reliability of day-to-day software is questionable. In fact, NIST estimated in 2002 that software failures cost the US economy alone about $59.5 billion every year, and that improvements in software testing infrastructure might save one-third of this cost.

Testing remains the primary way to improve reliability of software. Billions of dollars are spent on testing in the software industry, as testing usually accounts for more than half the cost of software development. In spite of its success in both commercial and open-source software, testing suffers from at least four limitations. First, the primary method of generating test inputs is manual. Automated test input generation techniques, which are less widely-used, have limitations: random testing can only detect a few bugs; symbolic execution based testing, which is more exhaustive, depends on automated theorem proving techniques and hence limited by the power of underlying theorem prover. Second, testing becomes notoriously hard for large concurrent programs due to the inherent non-determinism in such programs. Third, testing is ad hoc: the translation of the specification into program assertions is mostly done manually. Finally, testing can find bugs in a program; however, it cannot prove a program correct.

In this dissertation, we partly address these problems by developing systematic and automated testing methods for sequential and concurrent programs.

We first focus on sequential programs that can get data inputs from their environments. We assume that the behavior of such a program depends solely on the input that it gets from its external environment—for a given input the behavior of the program is deterministic. We have developed a novel method to systematically and automatically test sequential programs. The method is called

concolic testing. Concolic testing combines concrete and symbolic execution to generate test inputs that enable a program to explore all the distinct feasible execution paths of a programs at most once, while avoiding redundant test inputs as well as false warnings. Since concolic testing tries to explore all possible execution paths of a program, it can catch generic errors such as assertion violations, uncaught exceptions, segmentation faults, and so on.

Large real-world programs are almost invariably concurrent. In concurrent programs, several threads or actors or processes execute concurrently communicating with each other either through shared memory or message passing. Testing concurrent programs is notoriously hard because of the exponentially large number of possible interleavings of concurrent events that the execution of such programs generate. We have extended concolic testing with a new technique called *race-detection and flipping* which provides ways of dramatically reducing the exploration space for shared memory concurrent programs. In particular, the extended method uses the concrete execution of concolic testing to compute *the causality relation*, an abstract relation, between the events in a concurrent execution. Two executions of a concurrent program are said to be equivalent if they exhibit the same events and the events are related by the same causality relation. We use the computed causality relation to provide a novel technique for exploring non-equivalent (with respect to the causality relation) execution paths of a concurrent program. The extended testing method can catch concurrency related generic errors such as data races and deadlocks, in addition to generic errors such as assertion violations, uncaught exceptions, and segmentation faults. This extension provides the first technique to effectively and systematically test concurrent programs with complex data inputs. Because our testing method is designed to explore execution paths of a concurrent program, we term the method *Explicit Path Model Checking*.

Errors in a program may not be always generic: they may be due to the violation of the functional requirement of the program, for example, requirements that are provided in a formal functional specification of the program. To test a program against its formal specification, we combine concolic testing with *runtime monitoring*. Runtime monitors are small software units which are synthesized automatically from a formal specification for the program and weaved into the code to dynamically check if the specification is violated.

While applying concolic testing combined with runtime monitoring to concurrent multi-threaded

programs, we observed that exploring only non-equivalent execution paths is not sufficient for catching violations of temporal properties—a temporal property may be simultaneously satisfied and violated by two different equivalent execution paths. Therefore, to test a temporal specification against a program, we need to explore and monitor all possible execution paths of the program. Unfortunately, it has been shown that the number of possible execution paths of a concurrent program can be exponentially larger than the number of non-equivalent paths. This makes runtime monitoring of all execution paths of even a relatively simple concurrent program impractical.

To address this difficulty, we have developed a novel technique for predictive monitoring of concurrent programs. In this technique we generate, from an observed execution path, all its equivalent execution paths and represent them compactly as an abstract model called a *computation lattice*. We show that monitoring of temporal properties on this model can be done efficiently. Since this technique enables us to predict violations of properties in non-observed execution paths without re-executing the program, we call the technique *predictive monitoring*. Observe that predictive monitoring can predict and monitor all execution paths equivalent to a given execution path; we still need concolic testing extended with race-detection and flipping to explore all non-equivalent execution paths.

Based on the above methods we have developed tools for testing both C and Java programs. The testing tool for C is called CUTE; CUTE can only handle sequential C programs. The testing tool for Java is called jCUTE; jCUTE can test multi-threaded Java programs. Both CUTE and jCUTE can find generic errors. For Java, we have also implemented the predictive monitoring method in jCUTE. We have used CUTE and jCUTE to find bugs in several real-world software systems. CUTE found two previously unknown errors—a segmentation fault and an infinite loop—in SGLIB, a popular C data structure library used in a commercial software. Using jCUTE, we tested the thread-safe Java Collection framework provided with the Sun Microsystems' Java 1.4. Surprisingly, we discovered several data races, deadlocks, uncaught exceptions, and an infinite loop in this widely used library. All of these are concurrency related potential bugs. In addition, jCUTE found a previously known subtle time-partitioning error in the Honeywell's DEOS real-time operating system developed for use in small business aircraft. jCUTE detected well-known security attacks in a concurrent implementation of the Needham-Schroeder and the TMN protocols.

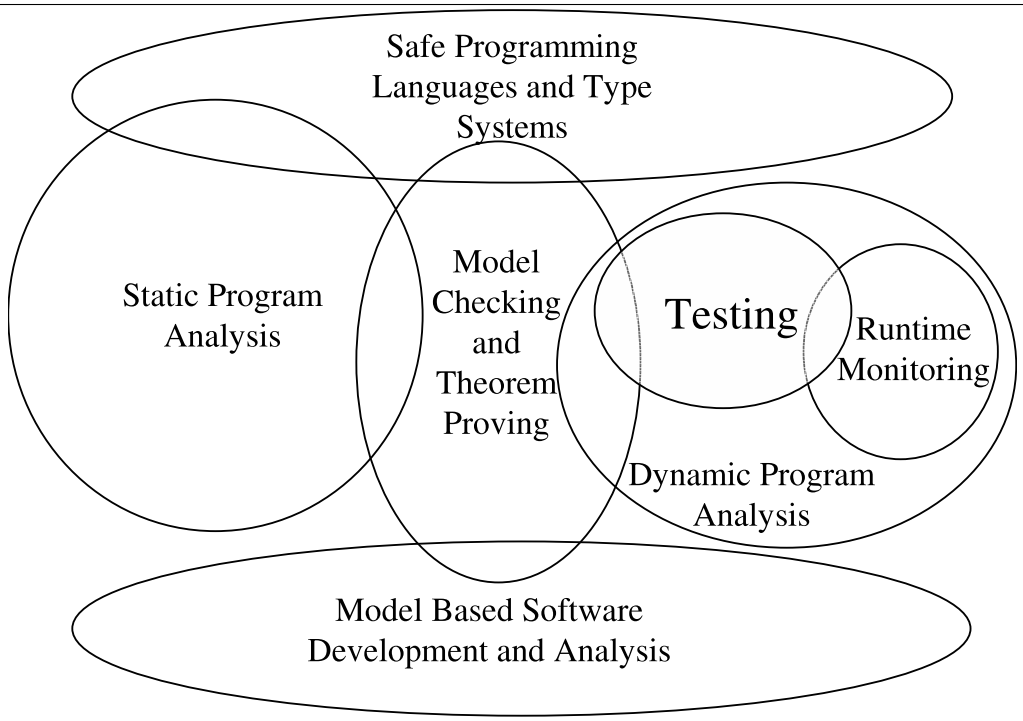

Figure 1.1: The 'Big Picture'

1.1 Background

We describe the 'big picture' and how this dissertation fits in the big picture. A large fraction of the research in software engineering and programming languages is devoted to the problem of building reliable software. The research can be broadly classified into five overlapping categories (see Figure 1.1) as follows.

- *Safe Programming Languages and Type Systems:* The goal of research in this area is to develop programs using programming constructs and type systems that ensure that the developed programs are free from certain classes of bugs such as memory errors, information leaks, data races, deadlocks, etc. The limitations of these approaches is that they cannot guarantee that a program is free from all kinds of bugs. Moreover, sophisticated type systems make programming a harder task.

- *Static Analysis:* The methods developed by research in this area help to find bugs in programs by statically analyzing the source code of a program. Methods in this area include automated theorem proving, pointer analysis, software model checking based on predicate abstraction

and refinement, and so on. Software reliability methods based on static analyses are often conservative, resulting in false notification of errors.

- *Dynamic Analysis:* Dynamic analysis relies on the runtime information obtained from an actual execution of a program. Various techniques based on dynamic program analyses include automated testing, runtime verification, explicit state model checking, and so on. Since dynamic analysis based methods detect bugs by observing the actual execution of a program, the bugs inferred are often real bugs. Unfortunately, since dynamic program analyses based methods analyze a single execution of program, they cannot discover all bugs.

- *Model Checking:* Model checking methods aim to prove that a program meets it formal specification. They include methods based on both static analyses and dynamic analyses. The main limitation of model checking methods is that they do not scale for large programs as the state space to be explored in such methods becomes enormous.

- *Model Based Software Development and Analysis:* The research in this area is based on the philosophy that the development of software should begin with the design of a formal model. Although this is a rigorous and systematic method of developing software, the cost and expertise required for developing software using this paradigm is often large. This often limits the applicability of the paradigm to safety critical systems.

The work presented in this dissertation falls under the category of dynamic program analysis. We believe in the philosophy that we can catch real bugs by actually executing a program. However, dynamic analysis means that we are restricted to information that is observed in an execution. We try to remove this limitation in a scalable way by directing a program to execute in all possible ways. Moreover, we develop methods that can, by observing a successful execution, predict bugs with certainty in other unobserved executions. In a broader sense, this thesis makes dynamic program analysis more systematic and rigorous; in other words, the thesis bridges some of the gap between model checking and testing.

1.2 Outline

The rest of the dissertation is organized as follows. In Chapter 2, we give an overview of our testing algorithms using simple examples. We use three different examples to introduce concolic testing, race-detection and flipping, and predictive monitoring.

In Chapter 3, to simplify the description of our testing algorithms, we introduce a simple imperative shared-memory multi-threaded programming language, called Scil. Most features of common high level imperative programming languages such as C and Java can be translated into Scil. We then formalize the execution paths of a program written in Scil in terms of a partial order relation called the causality relation and show how the causality relation can be tracked at runtime using dynamic vector clocks.

In Chapters 4, we describe our testing algorithm for the sequential fragment of Scil. The method is called concolic testing. We discuss the various advantages of concolic testing over traditional symbolic testing.

In Chapter 5, we extend concolic testing with race-detection and flipping to test a concurrent program written in Scil. We first describe a simple inefficient testing algorithm in which we use binary semaphores to effectively control the execution of threads. Then we modify the algorithm with the race-detection and flipping technique to improve the efficiency of testing concurrent programs. Finally, we suggest a further optimization of the testing algorithm.

In Chapter 6, we introduce predictive monitoring. We start this chapter by briefly introducing non-deterministic finite automata as monitors for temporal specifications. We then describe an algorithm for constructing an abstract computation model, called computation lattice, by observing a concurrent execution path. We show that monitoring of a non-deterministic finite automata on a computation lattice can be done efficiently and on-the-fly. Finally, we propose a heuristics, called causality cone heuristics, to make predictive monitoring more effective.

In Chapter 7, we describe the implementation of the testing methods in two tools CUTE and jCUTE for testing C and Java programs, respectively. We then report our experience on applying these tools on some real-world programs. In Chapter 8, we briefly review the literature related to the work in this dissertation.

Chapter 2

Overview

We give an overview of concolic testing, the race-detection and flipping algorithm, and predictive monitoring through a sequence of simple examples. The examples used to describe each of the three methods are different from each other. This is done to keep the examples simple and relevant to the methods they illustrate. We introduce concolic testing through a simple sequential program containing an error statement which may be executed for some input. The example is used to show how concolic testing generates test inputs one by one so that the program is directed to execute the error statement. The second example is a simple shared-memory multi-threaded program also containing an error statement. We use this example to describe how concolic testing is extended with the race-detection and flipping algorithm. The third example is another simple shared-memory multi-threaded program, which is required to obey a given temporal specification. The example illustrate the use of predictive monitoring to catch temporal specification violations.

2.1 Concolic Testing through an Example

In concolic testing, our goal is to generate data inputs that would exercise all the feasible execution paths of a sequential program. We first describe the essential idea behind concolic testing and then use an example to illustrate it.

Our algorithm for concolic testing uses concrete values as well as symbolic values for the inputs, and executes a program both concretely and symbolically. The symbolic execution is similar to the traditional symbolic execution [55], except that the algorithm follows the path that the concrete execution takes. During the course of the execution, it collects the constraints over the symbolic values at each branch point (i.e., the *symbolic constraints*). At the end of the execution, the algorithm has computed a sequence of symbolic constraints corresponding to each branch point.

We call the conjunction of these constraints a *path constraint*. Observe that all input values that satisfy a given path constraint will explore the same execution path.

The algorithm first generates a random input. Then the algorithm does the following in a loop: it executes the code with the generated input. At the same time the algorithm computes the symbolic constraints. It backtracks and generates a new input and executes the program again. The algorithm repeats these steps until it has explored all possible distinct execution paths using a depth first search strategy. The choice of a new input is made as follows: The algorithm picks a constraint from the symbolic constraints that were collected along the execution path and negates the constraint to define a new path constraint. The algorithm then finds, if possible, some concrete values that satisfy the new path constraint. These values are used as input for the next execution.

A complication arises from the fact that for some symbolic constraints, our constraint solver may not be powerful enough to compute concrete values that satisfy the constraints. To address this difficulty, such symbolic constraints *are simplified by replacing some of the symbolic values with concrete values.* [1]

We use a simple example to illustrate how our tool CUTE performs concolic testing. Consider the C function `testme` shown in Figure 2.1. The variables `p` and `x` in this function receive input from the external environment. (A program gets input using the expression `input`(). Observe that `input`() captures the various functions through which a program may receive data from its external environment.) Note that, `p` is a pointer, and thus the input includes the memory graph reachable from that pointer. In this example, the graph is a list of `cell` allocation units. The function `testme` has an error that can be reached given some specific values of the input.

For the example function `testme`, CUTE first non-randomly generates `NULL` for `p` and randomly generates 236 for `x`, respectively. Figure 2.1 shows this input to `testme`. As a result, the first execution of `testme` takes the `then` branch of the first `if` statement and the `else` branch of the second `if`. Let p_0 and x_0 be the symbolic values of `p` and `x`, respectively, at the beginning of the execution. CUTE collects the constraints from the predicates of the branches executed in this path: $x_0 > 0$ (for the `then` branch of the first `if`) and $p_0 = \texttt{NULL}$ (for the `else` branch of the second `if`).

[1] Because of this, our algorithm is complete only if given an oracle that can solve all constraints in a program, and the length and the number of paths is finite. Note that because the algorithm does concrete executions, it is sound, i.e. all bugs it infers are real.

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int
f(int v) {
  return 2*v + 1;
}

int testme() {
  cell * p = input();
  int x = input();
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          ERROR;
  return 0;
}
```

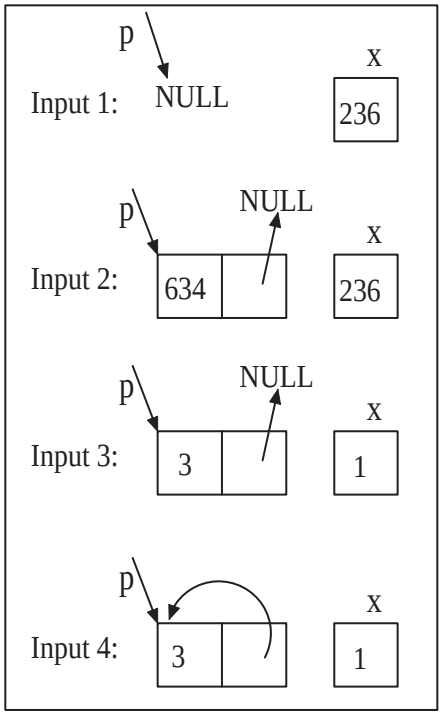

Figure 2.1: A Simple Sequential Program and the Inputs that CUTE Generates

The constraint sequence $\langle x_0 > 0, p_0 = \texttt{NULL}\rangle$ is called a *path constraint*.

CUTE next solves the path constraint $\langle x_0 > 0, p_0 \neq \texttt{NULL}\rangle$, obtained by negating the last constraint, to drive the next execution along an alternative path. The solution that CUTE proposes is $\{p_0 \mapsto \text{non-}\texttt{NULL}, x_0 \mapsto 236\}$ which requires that CUTE make `p` point to an allocated `cell` that introduces two new components, `p->v` and `p->next`, to the reachable graph. Accordingly, CUTE randomly generates 634 for `p->v` and non-randomly generates `NULL` for `p->next`, respectively, for the next execution. In the second execution, `testme` takes the `then` branch of the first and the second `if` statement and the `else` branch of the third `if` statement. For this execution, CUTE generates the path constraint $\langle x_0 > 0, p_0 \neq \texttt{NULL}, 2 \cdot x_0 + 1 \neq v_0\rangle$, where p_0, v_0, n_0, and x_0 are the symbolic values of `p`, `p->v`, `p->next`, and `x`, respectively. Note that CUTE computes the expression $2 \cdot x_0 + 1$ (corresponding to the execution of `f`) through an inter-procedural, dynamic tracking of symbolic expressions.

CUTE next solves the path constraint $\langle x_0 > 0, p_0 \neq \texttt{NULL}, 2 \cdot x_0 + 1 = v_0\rangle$, obtained by negating the last constraint and generates Input 3 from Figure 2.1 for the next execution. Note that the specific value of x_0 has changed, but the value remains in the same equivalence class with respect

to the predicate where it appears, namely $x_0 > 0$. On Input 3, `testme` takes the `then` branch of the first three `if` statements and the `else` branch of the fourth `if` statement. CUTE generates the path constraint $\langle x_0 > 0, p_0 \neq \texttt{NULL}, 2 \cdot x_0 + 1 = v_0, p_0 \neq n_0 \rangle$. This path constraint includes dynamically obtained constraints on pointers. CUTE handles constraints on pointers, but requires no static alias analysis. To drive the program along an alternative path in the next execution, CUTE solves the constraints $\langle x_0 > 0, p_0 \neq \texttt{NULL}, 2 \cdot x_0 + 1 = v_0, p_0 = n_0 \rangle$ and generates Input 4 from Figure 2.1. On this input, the fourth execution of `testme` reveals the error in the code.

2.2 The Race-Detection and Flipping Algorithm through an Example

For shared-memory multi-threaded programs, we extend concolic testing with the race-detection and flipping algorithm. In the extension, our goal is to generate thread schedules as well as data inputs that would exercise all non-equivalent executions paths of a shared-memory multi-threaded program. Apart from collecting symbolic constraints, the algorithm computes the race condition between various events in the execution of a program, where, informally, an event represents the execution of a statement in the program by a thread. We say that two events are in a *race* if they are the events of different threads, they access (i.e. read, write, lock, or unlock) the same memory location without holding a common lock, and the order of the happening of the events can be permuted by changing the schedule of the threads. The race conditions are computed by analyzing the concrete execution of the program with the help of dynamic vector clocks for multi-threaded programs introduced in Chapter 3.

The extended algorithm first generates a random input and a schedule which specifies the order of the execution of threads. Then the algorithm does the following in a loop: it executes the code with the generated input and the schedule. At the same time the algorithm computes the race conditions between various events as well as the symbolic constraints. It backtracks and generates a new schedule or a new input and executes the program again. It continues until it has explored all possible distinct execution paths using a depth-first search strategy. The choice of new inputs and schedules is made in one of the following two ways:

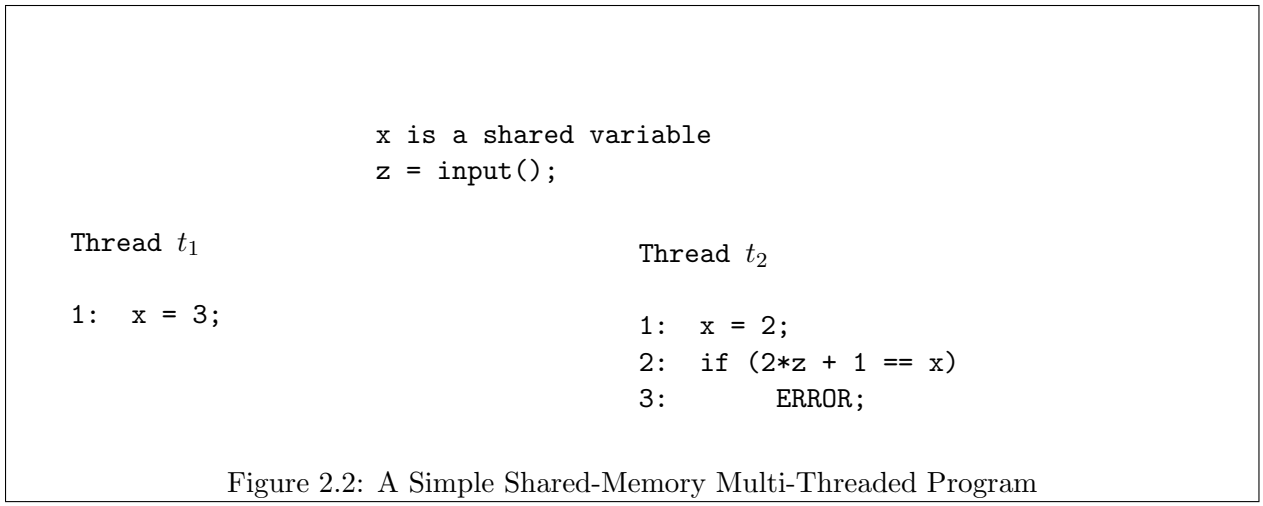

Figure 2.2: A Simple Shared-Memory Multi-Threaded Program

1. The algorithm picks a constraint from the symbolic constraints that were collected along the execution path and negates the constraint to define a new path constraint. The algorithm then finds, if possible, some concrete values that satisfy the new path constraint. These values are used as input for the next execution.

2. The algorithm picks two events which are in a race and generates a new schedule that at the point where the first event happened, the execution of the thread involved in the first event is *postponed* or *delayed* as much as possible. This ensures that the events involved in the race get *flipped* or re-ordered when the program is executed with the new schedule. The new schedule is used for the next execution.

We illustrate how jCUTE performs concolic testing along with race-detection and flipping using the sample program P in Figure 2.2. The program has two threads t_1 and t_2, a shared integer variable `x`, and an integer variable `z` which receives an input from the external environment at the beginning of the program. Each statement in the program is labeled. The program reaches the `ERROR` statement in thread t_2 if the input to the program is `1` (i.e., `z` gets the value `1`) and if the program executes the statements in the following order: $(t_2, 1)(t_1, 1)(t_2, 2)(t_2, 3)$, where each event, represented by a tuple of the form (t, l), in the sequence denotes that the thread t executes the statement labeled l.

jCUTE first generates a random input for `z` and executes P with a default schedule. Without loss of generality, the default schedule always picks the thread which is enabled and which has

the lowest index. Thus, the first execution of P is $(t_1, 1)(t_2, 1)(t_2, 2)$. Let z_0 be the symbolic value of `z` at the beginning of the execution. jCUTE collects the constraints from the predicates of the branches executed in this path. For this execution, jCUTE generates the path constraint $\langle 2 * z_0 + 1! = 2 \rangle$. jCUTE also decides that there is a race condition between the first and the second event because both the events access the same variable `x` in different threads without holding a common lock and one of the accesses is a write of `x`.

Following the depth-first search strategy, jCUTE picks the only constraint $2 * z_0 + 1! = 2$, negates it, and tries to solve the negated constraint $2 * z_0 + 1 = 2$. This has no solution. Therefore, jCUTE backtracks and generates a schedule such that the next execution becomes $(t_2, 1)(t_2, 2)(t_1, 1)$ (here the thread involved in the first event of the race in the previous execution is delayed as much as possible). This execution re-orders the events involved in the race in the previous execution.

During the above execution, jCUTE generates the path constraint $\langle 2 * z_0 + 1! = 2 \rangle$ and computes that there is a race between the second and the third events. Since the negated constraint $2 * z_0 + 1 = 2$ cannot be solved, jCUTE backtracks and generates a schedule such that the next execution becomes $(t_2, 1)(t_1, 1)(t_2, 2)$. This execution re-orders the events involved in the race in the previous execution.

In the above execution, jCUTE generates the path constraint $\langle 2 * z_0 + 1! = 3 \rangle$. jCUTE solves the negated constraint $2 * z_0 + 1 = 3$ to obtain $z_0 = 1$. In the next execution, it follows the same schedule as the previous execution. However, jCUTE starts the execution with the input variable `z` set to `1` which is the value of `z` that jCUTE computed by solving the constraint. The resultant execution becomes $(t_2, 1)(t_1, 1)(t_2, 2)(t_2, 3)$ which hits the `ERROR` statement of the program.

2.3 Predictive Monitoring through an Example

Concolic testing extended with the race-detection and flipping algorithm tries to explore all non-equivalent execution paths of a program. As such this method can catch generic errors such as assertion violations, uncaught exceptions, data races, and deadlocks. However, sometime we may also want to test a program against a formal specification, rather than hunting for generic errors. A trivial way to enable concolic testing to test a program against a formal specification would be to combine it with runtime monitoring. Unfortunately, if our program is a shared-

```
            Initially x = 0 and y = 0
            z = input();

Thread t1                                Thread t2

1:  if (z == 100) {                      1:  y = 2;
2:      x = 4;
3:  } else {
4:      x = 5;}
```

Figure 2.3: Another Simple Shared-Memory Multi-Threaded Program

memory multi-threaded program, and if our specification is some safety formula in a temporal logic, then the combination of concolic testing and runtime monitoring might miss violations of the specification. This is because a temporal safety property may be simultaneously satisfied and violated by two different execution paths that are equivalent. Since concolic testing extended with the race-detection and flipping algorithm explores non-equivalent execution paths, the combined method may miss violations of a temporal property.

To illustrate the above mentioned limitation, consider the simple multi-threaded program in Figure 2.3. The method of concolic testing extended with the race-detection and flipping algorithm would explore two non-equivalent execution paths of the program, namely $(t_1, 1)(t_1, 3)(t_1, 4)(t_2, 1)$ and $(t_1, 1)(t_1, 2)(t_2, 1)$. Since the program cannot exhibit any other non-equivalent execution path, concolic testing will stop after exploring the two paths.

Suppose we want to test the program against the temporal property ‘‘`Always x is greater than equal to y,`’’ also written as $\Box(\texttt{x} \geq \texttt{y})$ in linear temporal logic. Both the executions explored by concolic testing do not violate the property because none of the executions reach a state where `x` is less than `y`. However, there exists two other execution paths, namely $(t_2, 1)(t_1, 1)(t_1, 3)(t_1, 4)$ and $(t_2, 1)(t_1, 1)(t_1, 2)$ each of which violates the property. Unfortunately, concolic testing would not explore these paths as each one of them is equivalent to some already explored path.

One way to address this problem is to generate a runtime monitor which simply checks if `x` is greater than `y`. The monitor is then inserted into the code through instrumentation so that whenever `x` or `y` is updated the monitor check is invoked. The instrumented program is given in Figure 2.4.

```
            Initially x = 0 and y = 0
            z = input();

Thread t1                                   Thread t2

1:  if (z == 100){                          1:  y = 2;
2:      x = 4;                              2:  assert(x >= y);
3:      assert(x >= y);
4:  } else {
5:      x = 5;
6:      assert(x >= y);}
```

Figure 2.4: Instrumented Program

If we apply concolic testing to the instrumented program in Figure 2.4, eight distinct execution paths of the program will be explored and a violation of the property will be reported. Note that this simple solution results in the exploration of considerably large number of execution paths compared to the number of paths explored by concolic testing of the original program.

We have developed predictive monitoring to test shared-memory multi-threaded programs efficiently. In predictive monitoring for each execution path explored by concolic testing, we infer other equivalent execution paths statically without re-executing the program for each such equivalent path. All the equivalent execution paths corresponding to an observed execution path are then monitored against the temporal property efficiently.

For example, consider the execution path $(t_1, 1)(t_1, 3)(t_1, 4)(t_2, 1)$ explored by concolic testing of the original program. Since there is no causal connection between any event of the thread t_1 and the event of the thread t_2, we can permute the event from the thread t_2 with any event from thread t_1. This allows us to construct an alternate feasible execution path $(t_2, 1)(t_1, 1)(t_1, 3)(t_1, 4)$ equivalent to the observed execution path. This alternate path violates the temporal property. Thus by combining concolic testing with predictive monitoring, we have detected the violation of the temporal property by executing the program exactly twice.

Chapter 3

Programming and Execution Model

We introduce a simple shared-memory multi-threaded programming language called Scil. In the subsequent chapters, we use this programming language to describe our testing algorithms. To simplify the description of our testing algorithms, we keep Scil free from function calls, function definitions, and other high-level programming features. In Chapter 7, we briefly discuss how we handle function calls found in general imperative languages. In the multi-threaded fragment of Scil, we do not include concurrency primitives such as *wait*, *notify*, or *join*. This is again done to simplify the exposition. Our implementation for Java handles all these primitives. In Section 3.2, we define a partial order relation, called the *causality relation*, to abstract the concurrent execution of a program. We describe a novel dynamic vector clock algorithm that keeps track of this causality relation at runtime. We use the partial order abstraction of a concurrent execution to describe the race-flipping and detection algorithm in Chapter 5 and the predictive monitoring algorithm in Chapter 6. Finally, in Section 3.3, we use the execution model to formally state the problems that we solve in the subsequent chapters.

3.1 Programming Model

In order to simplify the description of our testing methods, we define a simple shared-memory multi-threaded imperative language, Scil (Figure 3.1). A Scil program is a set of threads that are executed concurrently, where each thread executes a sequence of statements. Note that each statement in a program is labeled. Threads in a program communicate by acquiring and releasing locks and by accessing (i.e., by reading or writing) shared memory locations.

A Scil program may receive data inputs from its environment. Observe that the availability of an input earlier than its use does not affect an execution. Without loss of generality, we assume that

$$
\begin{array}{rl}
P & ::= Stmt^* \\
Stmt & ::= l\colon S \\
S & ::= v \leftarrow lv \mid *lv \leftarrow lv \mid lv \leftarrow e \mid \texttt{if}\ \ p\ \ \texttt{goto}\ \ l' \\
 & \quad \mid \mathit{fork}(l) \mid \mathit{lock}(\&v) \mid \mathit{unlock}(\&v) \mid \texttt{START} \mid \texttt{HALT} \mid \texttt{ERROR} \\
e & ::= v \mid \&v \mid *lv \mid c \mid lv \mid lv\ \ op\ \ lv \mid \mathit{input}() \\
 & \quad \text{where } op \in \{+, -, /, *, \%, \ldots\}, v \text{ is a shared variable,} \\
 & \quad lv \text{ is a variable local to a thread, } c \text{ is a constant} \\
p & ::= lv = lv \mid lv \neq lv \mid lv < lv \mid lv \leq lv \mid lv \geq lv \mid lv > lv
\end{array}
$$

Figure 3.1: Syntax of Scil

all such inputs are available from the beginning of an execution; again this assumption simplifies the description of our algorithm.

We now informally describe the semantics of Scil. Consider a Scil program P. The execution of a thread terminates when it executes a `HALT` or an `ERROR` statement. `START` represents the first statement of a program under test. CUTE uses the CIL framework [68] to convert more complex statements (with no function calls) into this simplified form by introducing temporary variables. Some examples of converting complex code snippets into Scil code[1] is given in Table 3.1. Details of handling of function calls using a symbolic stack are discussed in Section 7.1.

During the execution of a Scil program, a single thread, namely t_{main}, starts by executing the first statement of the program. This thread t_{main} is comparable to the `main` thread in Java. The initial thread t_{main} or any subsequently created thread in the program can create new threads by calling the statement $\mathit{fork}(l)$, where l is the label of the statement to be executed by the newly created thread of the program.

A Scil program gets input using the expression $\mathit{input}()$. Observe that $\mathit{input}()$ captures the various functions through which a program in Java may receive data from its external environment.

A program may have two kinds of variables: variables local to a thread (denoted by lv) and variables shared among threads (denoted by v). The expression $\&v$ denotes the address of the variable v, and $*v$ denotes the value of the address stored in v. Note that associated with each address is a value that is either a *primitive* value or another memory address (i.e., *pointer*) and a

[1] In the converted code, we do not label every statement as required by Scil. This is done to keep the converted code relatively clean.

Original Code	Transformed Code

`**v = 3;`	`t1 = *v;  //t1 is a new variable`
	`*t1 = 3;`

`p[i] = q[j];`	`t1 = p + i;  //t1 is a new variable`
	`t2 = q + j;  //t2 is a new variable`
	`*t2 = *t1;`

`assert(x > 0);`
`x = 10;`
	`if (x > 10) goto L;`
	`ERROR;`
	`L: x = 10;`

`if (x > 0) {`
`  x = 1;`
`} else {`
`  x = -1;`
`}`
`y = x + y;`
	`if (x <= 0) goto L1;`
	`x = 1;`
	`if (true) goto L2;`
	`L1: x = -1;`
	`L2: y = x + y;`

`switch (c) {`
`  case 1:`
`  x = x + 1;`
`  break;`
`  case 2:`
`  x = x - 1;`
`  break;`
`  default:`
`  x = 0;`
`  break;`
`}`
`x = x + c;`
	`if (c != 1) goto L1;`
	`x = x + 1;`
	`if (true) goto L3;`
	`L1: if (c != 2) goto L2;`
	`x = x - 1;`
	`if (true) goto L3;`
	`L2: x =0;`
	`L3: x = x + c;`

`sum = i = 0;`
`while (i < 10) {`
`  sum = sum + i;`
`  i++;`
`}`
`i = sum;`
	`sum = 0;`
	`i = 0;`
	`L1: if (i >= 10) goto L2;`
	`sum = sum + i;`
	`i = i + 1;`
	`if (true) goto L1;`
	`L2: i = sum;`

Table 3.1: Examples of Converting Code Snippets Involving High-Level Programming Constructs to SCIL

```
execute_program(P)
    while there is an enabled thread
        t_current = Non-deterministically pick a thread from the set of enabled threads;
        execute the next statement of thread t_current;
    if there is an active thread
        print "Found deadlock";
```

Figure 3.2: Default Scheduler for SCIL

given statement can have at most one shared variable access (i.e. read, write, lock, or unlock).

A program supports mutual exclusion by using locks: *lock*(&v) denotes the acquisition of the lock on the shared variable v and *unlock*(&v) denotes the release of the same. A thread suspends its execution if it tries to acquire a lock which is already acquired by another thread. Normal execution of the thread resumes when the lock is released by the other thread. We assume that the acquire and release of locks take place in a nested fashion as in Java. Locks are assumed to be *re-entrant*: if a thread already holds a lock on a shared variable, then an acquire of the lock on the same variable by the same thread does not deadlock. When a thread executes `HALT` or `ERROR`, all the locks held by the thread are released. For technical simplicity, we assume that the set of memory locations that can be locked or unlocked is disjoint from the set of memory locations that can be read or write.

The semantics of a program in the language is given using a scheduler. The scheduler runs in a loop (see Figure 3.2). We use the term *schedule* to refer to the sequence of choices of threads made by the scheduler during an execution. We assume that each execution of a program under test terminates.

On executing a statement *lock*(&v), a thread waits if the lock v is already held by another thread. Otherwise, the thread acquires the lock and continues its execution. A lock v already held by a thread t is released when t executes a statement of the form *unlock*(&v). Initially, the thread t_{main} is enabled. A thread is said to be *active* if it has been created and it has not already executed a `HALT` or an `ERROR` statement. A thread is said to be *enabled* if it is active and it is not waiting to acquire a lock.

The execution of a statement of the form *fork*(l) creates a new thread, makes it active, and sets

the program counter of the newly created thread to l. The loop of the scheduler terminates when the set of enabled threads is empty. The termination of the scheduler indicates either the normal termination of a program execution when the set of active threads is empty, or a deadlock state when the set of active threads is non-empty.

3.2 Execution Model

Let us consider a program P. The execution of each statement in P is an event. Note that a statement may involve access to a shared memory location. We represent an event as (t, l, a), where l is the label of the statement executed by thread t and a is the type of shared memory access in the statement. If the execution of the statement accesses a shared memory location, then $a = \mathbf{r}$ if the access is a read, $a = \mathbf{w}$ if the access is a write, $a = \mathbf{l}$ if the access is a lock, and $a = \mathbf{u}$ if the access is an unlock; otherwise, $a = \perp$. If the execution of a fork statement labeled l by a thread t creates a new thread t', then we get two events: $(t, l, \perp)$ representing the fork event on the thread t and $(t', \perp, \perp)$ representing the creation of the new thread. Thus the event $(t', \perp, \perp)$ represents the first event of any newly created thread t'. We use the term *access* to represent a read, a write, a lock, or an unlock of a shared memory location. We use the term *update* to represent a write, a lock, or an unlock of a shared memory location. We call an event

- a *fork event*, if the event is of the form $(t, l, \perp)$ and l is the label of a fork statement,
- a *new thread event*, if the event is of the form $(t, \perp, \perp)$,
- a *read*, a *write*, a *lock*, an *unlock*, an *access*, or an *update event*, if the event reads, writes, locks, unlocks, accesses, or updates a memory location, respectively,
- an *internal event*, if the event is not a fork event, a new thread event, or an access event.

An execution of P can be seen as a *sequence of events*. We call such a sequence an *execution path*. Note that the execution of P on several inputs may result in the same execution path. Let $\texttt{Ex}(P)$ be the set of all feasible execution paths exhibited by the program P on all possible inputs and all possible choices by the scheduler.

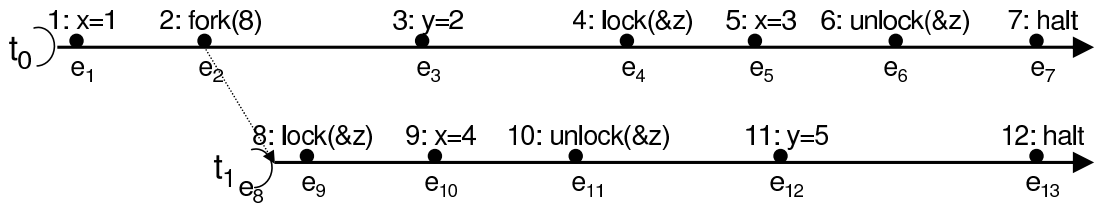

Figure 3.3: Time increases from left to right. $e_3 \parallel e_{10}$, $e_9 \lessdot e_4$, $e_{10} \not\lessdot e_5$, $e_3 \lessdot e_{12}$, $e_1 \not\lessdot e_{10}$, $e_1 \preccurlyeq e_{10}$, $e_1 \preccurlyeq e_{12}$, $e_1 \lhd e_9$, $e_3 \lhd e_4$, $e_3 \Updownarrow e_{12}$, etc.

If we view each event in an execution path as a node, then $\texttt{Ex}(P)$ can be seen as a tree. Such a tree is called the *computation tree* of a program. The goal of our testing method for concurrent programs is to systematically explore a minimum possible subset of the execution paths of $\texttt{Ex}(P)$ such that if a statement of P is reachable by a thread for some input and some schedule, the subset must contain an execution path in which that statement is executed. To achieve this, we abstract an execution path in terms of a partial order relation called *causal relation*. Any partial order represents a set of equivalent execution paths. In our testing algorithm, the goal is to exactly explore one execution path corresponding to each partial order. However, in the actual algorithm, we are able to guarantee that at least one—*not* at most one—execution path corresponding to each partial order is explored if a program has no data input (see Chapter 5 for a discussion on the effect of data input). We next define the various binary relations that we use to define a partial order.

In an execution path $\tau \in \texttt{Ex}(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *sequentially related* (denoted by $e \lhd e'$) iff:

1. $e = e'$, or
2. $t_i = t_j$ and e appears before e' in τ, or
3. $t_i \neq t_j$, t_i created the thread t_j, and e appears before e'' in τ, where e'' is the fork event on t_i creating the thread t_j, or
4. there exists an event e'' in τ such that $e \lhd e''$ and $e'' \lhd e'$.

Thus $\lhd$ is a partial order relation. We say $e \Updownarrow e'$ iff $e \not\lhd e'$ and $e' \not\lhd e$.

In an execution path $\tau \in \texttt{Ex}(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *shared-memory access precedence related* (denoted by $e <_m e'$) iff:

1. e appears before e' in τ, and

2. e and e' both access the same memory location m, and

3. one of them is an update of m.

In the above definition, it is worth remembering that the memory locations that can be locked or unlocked are disjoint from the memory locations that can be read or write. Therefore, if $e <_m e'$ and e (or e') is a lock or unlock of m, then the e' (or e) is also a lock or unlock of m. Similarly, if $e <_m e'$ and e (or e') is a write of m, then the e' (or e) is a read or write of m.

Given the definition of the sequential relation and the shared-memory access precedence relation, we can define another relation, called *causal relation*, as follows. In an execution path $\tau \in \texttt{Ex}(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *causally related* (denoted by $e \preccurlyeq e'$) iff:

1. $e \lhd e'$, or

2. $e <_m e'$ for some shared-memory location m, or

3. there exists e'' such that $e \preccurlyeq e''$ and $e'' \preccurlyeq e'$.

The causal relation is a partial-order relation. We say that $e \parallel e'$ iff $e \not\preccurlyeq e'$ and $e' \not\preccurlyeq e$. If $e \preccurlyeq e'$, then we say e *causally precedes* e'.

We next define a relation $\lessdot$, called *race relation*, that captures the race condition between two events. We say that any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ are *race related* (denoted by $e \lessdot e'$) iff

1. $e \Updownarrow e'$,

2. if e is a lock event and e'' is the corresponding unlock event, then $e'' <_m e'$ and there exists no e_1 such that $e_1 \neq e''$, $e_1 \neq e'$, $e'' \preccurlyeq e_1$, and $e_1 \preccurlyeq e'$, and

3. if e is a read or a write event, then $e <_m e'$ and there exists no e_1 such that $e_1 \neq e$, $e_1 \neq e'$, $e \preccurlyeq e_1$, and $e_1 \preccurlyeq e'$.

If two events in an execution path are related by $\lessdot$, then there exists an immediate *race* (data race or lock race) between the two events. Therefore, we call $\lessdot$ a *race* relation.

Figure 3.3 gives an example of the various relations defined above.

Given two execution paths τ and τ' in $\texttt{Ex}(P)$, we say that τ and τ' are *causally equivalent*, denoted by $\tau \equiv_{\preccurlyeq} \tau'$, iff τ and τ' have the same set of events and they are linearizations of the same $\preccurlyeq$ relation. We use $[\tau]_{\equiv_{\preccurlyeq}}$ to denote the set of all executions in $\texttt{Ex}$ that are equivalent to τ.

We define *a representative set of executions* $\texttt{REx} \subseteq \texttt{Ex}$ as the set that contains exactly one candidate from each equivalence class $[\tau]_{\equiv_{\preccurlyeq}}$ for all $\tau \in \texttt{Ex}$. Formally, $\texttt{REx}$ is a set such that following properties hold:

1. $\texttt{REx} \subseteq \texttt{Ex}$,
2. $\texttt{Ex} = \bigcup_{\tau \in \texttt{REx}} [\tau]_{\equiv_{\preccurlyeq}}$, and
3. for all $\tau, \tau' \in \texttt{REx}$, it is the case that $\tau \not\equiv_{\preccurlyeq} \tau'$.

The following result shows that a systematic and automatic exploration of each element in $\texttt{REx}$ is sufficient for testing.

Proposition 1. *If a statement is reachable in a program P for some input and schedule, then there exists a $\tau \in \texttt{REx}$ such that the statement is executed in τ.*

The proof of this proposition is straight-forward. If a statement is reachable then there exists an execution τ in $\texttt{Ex}$ such that the execution τ executes the statement. By the definition of $\equiv_{\preccurlyeq}$, any execution in $[\tau]_{\equiv_{\preccurlyeq}}$ executes the statement. Hence, the execution in $\texttt{REx}$ that is equivalent to τ executes the statement.

Dynamic Vector Clock

The causal relation between the events in an execution can be tracked efficiently at runtime using *dynamic vector clocks* (DVC). Dynamic vector clocks, which respect the fact that two reads can be permuted, extend the standard vector clocks [30] found in message passing systems. A dynamic vector clock $V \colon T \to \mathbb{N}$, where T is the set of threads that are present in the execution. We call such a map a *dynamic vector clock (DVC)* because its partiality reflects the intuition that

threads are dynamically created and destroyed. However, in order to simplify the exposition and the implementation, we represent each DVC V as a total map, where $V(t) = 0$ whenever V is not defined on thread t (i.e., if t has not been created).

We associate a DVC with every thread t and denote it by V_t. Moreover, we associate two DVCs V_m^a and V_m^w with every shared memory m; we call the former *access DVC* and the latter *update DVC*. For any two maps V and V', we say that $V \leq V'$ if and only if $V(t) \leq V'(t)$ for all $t \in T$. We say that $V \neq V'$ if and only if $V \not\leq V'$ and $V' \not\leq V$. $\max\{V, V'\}$ is the DVC with $\max\{V, V'\}(t) = \max\{V(t), V'(t)\}$ for each $t \in T$.

At the beginning of an execution, all vector clocks associated with threads and memory locations are empty. Whenever a thread t with current DVC V_t generates an event e, the following algorithm $\mathcal{A}$ is executed:

1. If e is not a fork event or a new thread event, then $V_t(t) \leftarrow V_t(t) + 1$.

2. If e is a read of a shared memory location m then

   $V_t \leftarrow \max\{V_t, V_m^w\}$

   $V_m^a \leftarrow \max\{V_m^a, V_t\}$

3. If e is a write, lock, or unlock of a shared memory location m then

   $V_m^w \leftarrow V_m^a \leftarrow V_t \leftarrow \max\{V_m^a, V_t\}$

4. If e is a fork event and if t' is the newly created thread then

   $V_{t'} \leftarrow V_t$

   $V_t(t) \leftarrow V_t(t) + 1$

   $V_{t'}(t') \leftarrow V_{t'}(t') + 1$

We call the algorithm $\mathcal{A}$ the *dynamic vector clock algorithm*. If e is an event of thread t, then we use $V\{e\}$ to denote the DVC of t after the event e, $V\{e\}_m^w$ to denote the DVC V_m^w after the event e, and $V\{e\}_m^a$ to denote the DVC V_m^a after the event e. If e is an event of thread t, then the event in thread t that happened immediately before e is denoted by $prev(e)$. Similarly, if e is an event of thread t, then the event in thread t that happened immediately after e is denoted by $next(e)$.

In an execution, if we update the DVCs according to $\mathcal{A}$, we want to show that the causality relation is tracked by the dynamic vector clocks, i.e., For any two events e and e', $e \preccurlyeq e'$ iff

$V\{e\} \leq V\{e'\}$. In order to prove this result, we first introduce some definitions and then state and prove the following four lemmas.

Let us fix an arbitrary but fixed execution path τ. Let E_t be the set of events of t in τ. Let T be the set of all threads created in τ. Let $E = \cup_{t \in T} E_t$. We use e_t^i to denote the i^{th} event of the thread t in $\mathcal{C}$. Given an event $e \in E$, we define $(e], (e]_t, (e]_m^a, (e]_m^w$ as follows.

- $(e] = \{e' \mid e' \in E \text{ and } e' \preccurlyeq e\}$,
- $(e]_t = E_t \cap (e]$,
- $(e]_m^a = \{e'' \mid e'' \preccurlyeq e' \text{and } e' \text{ is an access of } m \text{ and } e' \text{ is equal to or appears before } e \text{ in the sequence } \tau\}$, and
- $(e]_m^w = \{e'' \mid e'' \preccurlyeq e' \text{and } e' \text{ is an update of } m \text{ and } e' \text{ is equal to or appears before } e \text{ in the sequence } \tau\}$.

For any $E' \subseteq E$, we use $|E'|_t$ to denote $|E' \cap E_t|$. We say that a set $E' \subseteq E$ is *a monotonic set* if and only if the following fact holds: if $e_t^k \in E'$, then $e_t^i \in E'$ for all $1 \leq i \leq k$.

Lemma 2. *Given an event e, $(e]$, $(e]_m^w$, and $(e]_m^a$ are monotonic sets.*

Proof. Follows from the definition of $(e]$, $(e]_m^w$, and $(e]_m^a$ and the fact that $e_t^i \preccurlyeq e_t^k$, for all $1 \leq i \leq k$. □

Lemma 3. *Given any two monotonic sets $E' \subseteq E$ and $E'' \subseteq E$ and a thread $t \in T$, $|E' \cup E''|_t = max(|E'|_t, |E''|_t)$.*

Proof. Let $\bigcup_{1 \leq i \leq k'} \{e_t^i\}$ be the set $E' \cap E_t$ and $\bigcup_{1 \leq i \leq k''} \{e_t^i\}$ be the set $E'' \cap E_t$. Then $(E' \cup E'') \cap E_t$ is the set $\bigcup_{1 \leq i \leq k} \{e_t^i\}$, where $k = max(k', k'')$. Since $|E'|_t = k'$, $|E''|_t = k''$, and $|E' \cup E''|_t = k$, we have $|E' \cup E''|_t = max(|E'|_t, |E''|_t)$. □

Lemma 4. *Let e_t^k be an event in τ and let $e_{t'}^l$ be the event that appears immediately before e_t^k in τ. Then the following holds.*

(1) If e_t^k is a read of a memory location m, then

(a) $(e_t^k] = (e_t^{k-1}] \cup \{e_t^k\} \cup (e_{t'}^l]_m^w$,

(b) $(e_t^k]_m^a = (e_t^k] \cup (e_{t'}^l]_m^a$,

(c) $(e_t^k]_{m'}^a = (e_{t'}^l]_{m'}^a$, *for* $m' \neq m$,

(d) $(e_t^k]_{m'}^w = (e_{t'}^l]_{m'}^w$, *for any* m'.

(2) If e_t^k *is a write, lock, or unlock of a memory location* m*, then*

(a) $(e_t^k] = (e_t^{k-1}] \cup \{e_t^k\} \cup (e_{t'}^l]_m^a$,

(b) $(e_t^k]_m^a = (e_t^k]$,

(c) $(e_t^k]_m^w = (e_t^k]$,

(d) $(e_t^k]_{m'}^a = (e_{t'}^l]_{m'}^a$, *for* $m' \neq m$,

(e) $(e_t^k]_{m'}^w = (e_{t'}^l]_{m'}^w$, *for* $m' \neq m$.

(3) If e_t^k *is an internal event, then*

(a) $(e_t^k] = (e_{t-1}^k] \cup \{e_t^k\}$,

(b) $(e_t^k]_{m'}^a = (e_{t'}^l]_{m'}^a$, *for any* m',

(c) $(e_t^k]_{m'}^w = (e_{t'}^l]_{m'}^w$, *for any* m'.

(4) If e_t^k *is a fork event, then*

(a) $(e_t^k] = (e_t^{k-1}] \cup \{e_t^k\}$,

(b) $(e_t^k]_{m'}^a = (e_{t'}^l]_{m'}^a$, *for any* m',

(c) $(e_t^k]_{m'}^w = (e_{t'}^l]_{m'}^w$, *for any* m'.

(5) If e_t^k *is a new thread event, then*

(a) $(e_t^k] = (e_{t''}^{k'-1}] \cup \{e_t^k\}$, *where* $e_{t''}^{k'}$ *is the fork event that created the thread* t,

(b) $(e_t^k]_{m'}^a = (e_{t'}^l]_{m'}^a$, *for any* m',

(c) $(e_t^k]_{m'}^w = (e_{t'}^l]_{m'}^w$, *for any* m'.

Proof. (1) (a) Let $E' = (e_t^{k-1}] \cup \{e_t^k\} \cup (e_{t'}^l]_m^w$. We want to prove $(e_t^k] \subseteq E'$. Let $e \in (e_t^k]$. Since e_t^k is read of m, by the definition of $\preccurlyeq$ one of the following must hold:

- $e = e_t^k$. In this case $e \in \{e_t^k\} \subseteq E'$.

- $e \preccurlyeq e_t^{k-1}$. In this case $e \in (e_t^{k-1}] \subseteq E'$.
- $e \preccurlyeq e'$, where $e' <_m e_t^k$. By the definition of $<_m$, e' is a write of m and e' appears before or equals to $e_{t'}^l$. This implies $(e'] \subseteq (e_{t'}^l]_m^w$. Therefore, $e \in (e_{t'}^l]_m^w \subseteq E'$.

Hence, $(e_t^k] \subseteq E'$.

We want to prove $E' \subseteq (e_t^k]$. Let $e \in E'$. Then one of the following must hold:

- $e \in (e_t^{k-1}]$. Since $e_t^{k-1} \preccurlyeq e_t^k$, $(e_t^{k-1}] \subseteq (e_t^k]$. Therefore, $e \in (e_t^k]$.
- $e = e_t^k$. In this case, $e \in (e_t^k]$.
- $e \in (e_{t'}^l]_m^w$. Then by the definition of $(e_{t'}^l]_m^w$, there is a e' such that e' is a write[2] of m, $(e'] \subseteq (e_{t'}^l]_m^w$, and $e \preccurlyeq e'$. Since e_t^k is a read of m, $e' <_m e_t^k$. This implies that $e' \preccurlyeq e_t^k$ which implies $e \in (e_t^k]$.

Hence, $E' \subseteq (e_t^k]$.

(b) Let $E' = (e_t^k] \cup (e_{t'}^l]_m^a$. We want to prove that $(e_t^k]_m^a \subseteq E'$. Let $e \in (e_t^k]_m^a$. Then there are two cases:

- $e \preccurlyeq e_t^k$, in this case $e \in (e_t^k]$, or
- $e \preccurlyeq e'$, where e' is an access of m and appears before e_t^k. This implies $e' \in (e_{t'}^l]_m^a$. Therefore, $e \in (e_{t'}^l]_m^a$.

Hence, $(e_t^k]_m^a \subseteq E'$.

We want to prove that $E' \subseteq (e_t^k]_m^a$. Let $e \in E'$. Then one of the following must hold:

- $e \in (e_t^k]$. Since e_t^k is a read of m, $(e_t^k] \subseteq (e_t^k]_m^a$. Therefore, $e \in (e_t^k]_m^a$.
- $e \in (e_{t'}^l]_m^a$. Since $e_{t'}^l$ appears before e_t^k, $(e_{t'}^l]_m^a \subseteq (e_t^k]_m^a$. Therefore, $e \in (e_t^k]_m^a$.

Hence $E' \subseteq (e_t^k]_m^a$.

(c) Since e_t^k is not accessing m', $(e_t^k]_{m'}^a = (e_{t'}^l]_{m'}^a$ follows from the definitions of $(e_t^k]_{m'}^a$ and $(e_{t'}^l]_{m'}^a$.

(d) Since e_t^k is not updating m', $(e_t^k]_{m'}^w = (e_{t'}^l]_{m'}^w$ follows from the definitions of $(e_t^k]_{m'}^w$ and $(e_{t'}^l]_{m'}^w$.

[2]Since e_t^k reads m, m can only be read or write in τ. This is because we assume that the memory locations that can be locked or unlocked are disjoint from the memory locations that can be read or write.

(2) (a) Let $E' = (e_t^{k-1}] \cup \{e_t^k\} \cup (e_{t'}^l]_m^a$. We want to prove $(e_t^k] \subseteq E'$. Let $e \in (e_t^k]$. Since e_t^k is an update of m, by the definition of $\preccurlyeq$ one of the following must hold:

- $e = e_t^k$. In this case $e \in \{e_t^k\} \subseteq E'$.
- $e \preccurlyeq e_t^{k-1}$. In this case $e \in (e_t^{k-1}] \subseteq E'$.
- $e \preccurlyeq e'$, where $e' <_m e_t^k$. By the definition of $<_m$, e' is an access of m and e' appears before or equals to $e_{t'}^l$. This implies $(e'] \subseteq (e_{t'}^l]_m^a$. Therefore, $e \in (e_{t'}^l]_m^a \subseteq E'$.

Hence, $(e_t^k] \subseteq E'$.

We want to prove $E' \subseteq (e_t^k]$. Let $e \in E'$. Then one of the following must hold:

- $e \in (e_t^{k-1}]$. Since $e_t^{k-1} \preccurlyeq e_t^k$, $(e_t^{k-1}] \subseteq (e_t^k]$. Therefore, $e \in (e_t^k]$.
- $e = e_t^k$. In this case, $e \in (e_t^k]$.
- $e \in (e_{t'}^l]_m^a$. Then by the definition of $(e_{t'}^l]_m^a$, there is a e' such that e' is an access of m, $(e'] \subseteq (e_{t'}^l]_m^a$, and $e \preccurlyeq e'$. Since e_t^k is an update of m, $e' <_m e_t^k$. This implies that $e' \preccurlyeq e_t^k$ which implies $e \in (e_t^k]$.

Hence, $E' \subseteq (e_t^k]$.

(b) We want to prove $(e_t^k]_m^a = (e_t^k]$. Let $e \in (e_t^k]_m^a$. Therefore, $e \preccurlyeq e'$ where e' accesses m and appears before or equals to e_t^k. Since e_t^k is an update of m, by the definition of $<_m$, $e' <_m e_t^k$. Therefore, $e \preccurlyeq e_t^k$ or $e \in (e_t^k]$.

Let $e \in (e_t^k]$. Since e_t^k is an update of m, by the definition of $(e_t^k]_t^a$, $(e_t^k] \subseteq (e_t^k]_m^a$. Therefore, $e \in (e_t^k]$.

(c) We want to prove $(e_t^k]_m^w = (e_t^k]$. Let $e \in (e_t^k]_m^w$. Therefore, $e \preccurlyeq e'$ where e' updates m and appears before or equals to e_t^k. Since e_t^k is an update of m, by the definition of $<_m$, $e' <_m e_t^k$. Therefore, $e \preccurlyeq e_t^k$ or $e \in (e_t^k]$.

Let $e \in (e_t^k]$. Since e_t^k is an update of m, by the definition of $(e_t^k]_t^w$, $(e_t^k] \subseteq (e_t^k]_m^w$. Therefore, $e \in (e_t^k]$.

(d) Since e_t^k is not accessing m', $(e_t^k]_{m'}^a = (e_{t'}^l]_{m'}^a$ follows from the definitions of $(e_t^k]_{m'}^a$ and $(e_{t'}^l]_{m'}^a$.

(e) Since e_t^k is not updating m', $(e_t^k]_{m'}^w = (e_{t'}^l]_{m'}^w$ follows from the definitions of $(e_t^k]_{m'}^w$ and $(e_{t'}^l]_{m'}^w$.

(3) (a) Since each $(e^k_{t-1}] \subseteq (e^k_t]$ and $\{e^k_t\} \subseteq (e^k_t]$, $(e^k_{t-1}] \cup \{e^k_t\} \subseteq (e^k_t]$. Let $e \in (e^k_t]$. Since e^k_t accesses no shared memory location, there are two cases to consider here: $e = e^k_t$ or $e \preccurlyeq e^k_{t-1}$. In either case, $e \in (e^k_{t-1}] \cup \{e^k_t\}$. Hence, $(e^k_t] \subseteq (e^k_{t-1}] \cup \{e^k_t\}$.

(b) Since e^k_t is not accessing m', $(e^k_t]^a_{m'} = (e^l_{t'}]^a_{m'}$ follows from the definitions of $(e^k_t]^a_{m'}$ and $(e^l_{t'}]^a_{m'}$.

(c) Since e^k_t is not updating m', $(e^k_t]^w_{m'} = (e^l_{t'}]^w_{m'}$ follows from the definitions of $(e^k_t]^w_{m'}$ and $(e^l_{t'}]^w_{m'}$.

(4) The proof is similar to the previous case.

(5) (a) Let $E' = (e^{k'-1}_{t''}] \cup \{e^k_t\}$. We want to prove $(e^k_t] \subseteq E'$. Let $e \in (e^k_t]$. By the definition of $\preccurlyeq$ one of the following must hold:

- $e = e^k_t$. This implies that $e \in \{e^k_t\} \subseteq E'$.
- $e \preccurlyeq e'$, where e' is any event of t'' and e' appears before $e^{k'}_{t''}$. Therefore, $e' = e^i_{t''}$ for $1 \leq i \leq k' - 1$. This implies that $e \preccurlyeq e^{k'-1}_{t''}$ or $e \in (e^{k'-1}_{t''}]$.

Hence, $(e^k_t] \subseteq (e^{k'-1}_{t''}] \cup \{e^k_t\}$.

We want to prove $E' \subseteq (e^k_t]$. Let $e \in E'$. Then one of the following must hold:

- $e = e^k_t$. Therefore, $e \in (e^k_t]$.
- $e \in (e^{k'-1}_{t''}]$. In this case $(e^{k'-1}_{t''}] \lhd e^k_t$, $e \preccurlyeq (e^k_t]$.

Hence, $(e^{k'-1}_{t''}] \cup \{e^k_t\} \subseteq (e^k_t]$.

(b) Since e^k_t is not accessing m', $(e^k_t]^a_{m'} = (e^l_{t'}]^a_{m'}$ follows from the definitions of $(e^k_t]^a_{m'}$ and $(e^l_{t'}]^a_{m'}$.

(c) Since e^k_t is not updating m', $(e^k_t]^w_{m'} = (e^l_{t'}]^w_{m'}$ follows from the definitions of $(e^k_t]^w_{m'}$ and $(e^l_{t'}]^w_{m'}$.

□

Lemma 5. *For any event e in τ, for any $t \in T$, and for any shared memory location m, the following holds:*

(1) $V\{e\}(t) = |(e]|_t$,

(2) $V\{e\}^w_m(t) = |(e]^w_m|_t$, *and*

(3) $V\{e\}^a_m(t) = |(e]^a_m|_t$.

Proof. Let $\tau = e_1 e_2 \dots e_n$. We prove this by induction. The lemma clearly holds for e_1. Let us assume that the lemma holds for all e_i, where $1 \leq i \leq p < n$. We need to show that it also holds for e_{p+1}. The equalities $(1), (2)$, and (3) follows from the various cases of Lemma 4 and the algorithm $\mathcal{A}$. In the following we show it for one case. The equalities for the remaining cases can be derived in a similar way. Let $e_{p+1} = e^k_t$ and e^k_t be a read event of m. By $(1)(a)$ of Lemma 4, for any $t'' \in T$, $|(e^k_t]|_{t''} = |(e^{k-1}_t] \cup \{e^k_t\} \cup (e^l_{t'}]^w_m|_{t''}$. Since $e^k_t \notin (e^{k-1}_t]$ and $e^k_t \notin (e^l_{t'}]^w_m$, we have $|(e^k_t]|_{t''} = |(e^{k-1}_t] \cup (e^l_{t'}]^w_m|_{t''} + 1$. Therefore, by Lemma 2 and Lemma 3, we have $|(e^k_t]|_{t''} = max(|(e^{k-1}_t]|_{t''}, |(e^l_{t'}]^w_m|_{t''}) + 1 = max(V\{e^{k-1}_t\}(t''), V\{e^l_{t'}\}^w_m(t'')) + 1$. By the algorithm $\mathcal{A}$, $V\{e^k_t\}(t'') = max(V\{e^{k-1}_t\}(t''), V\{e^l_{t'}\}^w_m(t'')) + 1$. Therefore, $V\{e^k_t\}(t'') = |(e^k_t]|_{t''}$. □

Now we are ready to prove the following theorem.

Theorem 6. *For any two events* e *and* e', $e \preccurlyeq e'$ *iff* $V\{e\} \leq V\{e'\}$.

Proof. Let us assume that $e \preccurlyeq e'$. This implies that $(e] \subseteq (e']$. Since each of $(e]$ and $(e']$ are monotonic sets, for all $t \in T$, we have $(e]_t \subseteq (e']_t$. This implies $|(e]|_t \leq |(e']|_t$. Therefore, by (1) of Lemma 5, we have $V\{e\}(t) \leq V\{e'\}(t)$, for all $t \in T$. In other words, $V\{e\} \leq V\{e'\}$. □

Sequential Vector Clock

The sequential relation between the events in an execution can be tracked efficiently at runtime using *sequential vector clocks* (SVC). A sequential vector clock $VS\colon T \to \mathbb{N}$, where T is the set of threads that are present in the execution. We call such a map a *sequential vector clock (SVC).* Such a map can be partial because threads are created dynamically at runtime. To simplify the exposition and the implementation, we assume that each SVC VS is a total map, where $VS(t) = 0$ whenever VS is not defined on thread t.

We associate a SVC with every thread t and denote it by VS_t. At the beginning of an execution, all sequential vector clocks associated with threads are empty. Whenever a thread t with current SVC VS_t generates an event e, the following algorithm $\mathcal{A}_f$ is executed:

1. If e is not a fork event or a new thread event, then $VS_t(t) \leftarrow VS_t(t) + 1$.

2. If e is a fork event and if t' is the newly created thread then

   $VS_{t'} \leftarrow VS_t$

   $VS_t(t) \leftarrow VS_t(t) + 1$

   $VS_{t'}(t') \leftarrow VS_{t'}(t') + 1$

The algorithm $\mathcal{A}_f$ is called the *sequential vector clock* algorithm. We can associate a SVC with every event e, denoted by VS_e as follows. If e is executed by t and if VS_t is the vector clock of t just after the event e, then $VS_e = VS_t$.

In an execution, if we update the SVCs according to $\mathcal{A}_f$, then the following theorem holds:

Theorem 7. *For any two events e and e', $e \lhd e'$ iff $VS_e \leq VS_{e'}$.*

The proof of this theorem is a special case of the proof of Theorem 6, where we assume each access event as an internal event.

Theorem 8. *For any two event e and e', if the following holds:*

1. *$V\{e\} \neq V\{prev(e')\}$ given that $prev(e')$ exists, and*

2. *$V\{next(e)\} \neq V\{e'\}$ given that $next(e)$ exists, and*

3. *$V\{e\} \leq V\{e'\}$, and*

4. *$VS_e \neq VS_{e'}$.*

then

- *if each of e and e' is a read or a write event, then $e \lessdot e'$,*

- *if e is an unlock event and e' is a lock event, then $e'' \lessdot e'$, where e'' is the lock event corresponding to e.*

The proof follows from the definition of $\lessdot$, Theorem 6, and Theorem 7.

3.3 Results in Terms of the Execution Model

In Chapter 4, we describe concolic testing which tries to explore all paths in $\texttt{Ex}(P)$ for a sequential program P. Note that for a sequential program P, $\texttt{Ex}(P) = \texttt{REx}(P)$. However, for shared-memory multi-threaded programs, $\texttt{REx}(P) \subseteq \texttt{Ex}(P)$. In Chapter 5, we extend concolic testing with the race-detection and flipping algorithm. The extended method tries to explore all paths in a superset of $\texttt{REx}(P)$ and a small subset of $\texttt{Ex}(P)$. As such the method would not explore all the paths in an equivalence class $[\tau]_{\equiv_{\preccurlyeq}}$. However, if we have a formal temporal specification, then it is possible that one path in $[\tau]_{\equiv_{\preccurlyeq}}$ satisfies the specification and another path in $[\tau]_{\equiv_{\preccurlyeq}}$ violates the specification. Therefore, monitoring the execution paths explored by concolic testing extended with race-detection and flipping may not be sufficient to find a violation of the formal specification. To eliminate this limitation, we describe predictive monitoring in Chapter 6. Predictive monitoring enables us to use an execution path τ to monitor all paths in $[\tau]_{\equiv_{\preccurlyeq}}$ without re-executing each of these paths.

Chapter 4

Testing Sequential Programs

In this chapter, we present concolic testing,[1] a novel automatic and systematic technique for testing sequential programs. Concolic testing automates the *exploration of feasible execution paths* of sequential programs. Our technique repeatedly generates inputs that make the code under test execute different paths. The process continues until the code executes all different feasible paths (up to a given length). Our technique uses a symbolic execution; unlike the traditional testing techniques based on symbolic execution or static analysis [24, 118, 75, 115, 10, 14, 73, 22], our technique tightly couples the concrete and symbolic executions of the code under test. Specifically, our technique simultaneously runs both concrete and symbolic executions such that each of them gets feedback from the other. We call this technique *concolic testing*, as it uses a cooperative CONCrete and symbOLIC execution.

Our technique work as follows. Given a program to test, our technique generates concrete inputs one by one. After generating each input, the program is executed on that input simultaneously both concretely and symbolically. The symbolic execution follows the path taken by the concrete execution and maintains a symbolic state and generates symbolic path constraints. The information collected by the concrete-execution guided symbolic execution is then used to generate an input for the next execution that will lead the program through an execution path that was not explored before by the program. This process continues until all different feasible paths are executed exactly once.

We have implemented our technique in two publicly available tools, called CUTE (concolic Unit Testing Engine) and jCUTE, for testing C and Java programs respectively. jCUTE also implements the race-detection and flipping algorithm described in Chapter 5. In the rest of the

[1] This work is done partly in collaboration with Gul Agha, Patrice Godefroid, Nils Klarlund, and Darko Marinov. Part of this work appeared in [41, 87, 86].

chapter we will only refer to C programs and CUTE. Note the same principles are also applicable to sequential Java programs. For C programs, the basic testing unit can be either one function or a small number of related functions or the whole program. Such testing units can have pointer inputs, and thus the inputs consist not only of primitive values but of the whole memory graphs. The symbolic execution in CUTE builds constraints on pointer variables (*pointer constraints*) and primitive variables (*arithmetic constraints*), unlike some previous techniques that use only concrete values for pointers [41, 115, 118]. We have developed a novel solver for both arithmetic constraints and pointer constraints, which enables CUTE to generate memory graphs as inputs to the code under test. Our solver exploits the domain of this particular problem to efficiently generate inputs that are similar to the previous inputs. CUTE can also generate complex data structures, either through function sequences or from the code for data structure invariants (Section 4.2).

The rest of the chapter is organized as follows. In Section 4.1, we describe the various components of concolic testing. We show how to apply concolic testing to test data structures in Section 4.2. Finally, we conclude the chapter by discussing the advantages of concolic testing over the existing methods.

4.1 Concolic Testing

We describe the details of concolic testing for sequential programs written in SCIL. To restrict a SCIL program to be sequential, we disallow the use of the concurrency primitives $fork(l)$, $lock(v)$, $unlock(v)$. Because of this restriction, a SCIL program is always single-threaded and all the variables and memory locations are local to the initial thread t_{main}. The simplified syntax of a restricted SCIL program is given in Figure 4.1.

We first define the input logical input map that CUTE uses to represent inputs. We present how CUTE instruments programs and performs concolic execution. We then describe how CUTE solves the constraints after every execution. We finally discuss the approximations that CUTE uses for pointer constraints. In the next section, we present how CUTE handles complex data structures.

To explore execution paths, CUTE first instruments the code under test. CUTE then builds a *logical input map* $\mathcal{I}$ for the code under test. Such a logical input map can represent a memory

$$
\begin{array}{rl}
P & ::= Stmt^* \\
Stmt & ::= l\colon S \\
S & ::= v \leftarrow e \mid *v \leftarrow e \mid \texttt{if } p \texttt{ goto } l' \\
 & \quad \mid \texttt{START} \mid \texttt{HALT} \mid \texttt{ERROR} \\
e & ::= v \mid \&v \mid *v \mid c \mid v \ op \ v \mid input() \\
 & \quad \text{where } op \in \{+, -, /, *, \%, \ldots\}, v \text{ is a variable,} \\
 & \quad c \text{ is a constant} \\
p & ::= v = v \mid v \neq v \mid v < v \mid v \leq v \mid v \geq v \mid v > v
\end{array}
$$

Figure 4.1: Syntax of Sequential Scil

graph in a symbolic way. CUTE then repeatedly runs the instrumented code as follows:

1. It uses the logical input map $\mathcal{I}$ to generate a concrete input memory graph for the program and two symbolic states, one for pointer values and another for primitive values.
2. It runs the code on the concrete input graph, collecting constraints (in terms of the symbolic values in the symbolic state) that characterize the set of inputs that would take the same execution path as the current execution path.
3. It negates one of the collected constraints and solves the resulting constraint system to obtain a new logical input map $\mathcal{I}'$ that is similar to $\mathcal{I}$ but (likely) leads the execution through a different path. It then sets $\mathcal{I} = \mathcal{I}'$ and repeats the process.

4.1.1 Logical Input Map

CUTE keeps track of input memory graphs as a logical input map $\mathcal{I}$ that maps logical addresses to values that are either logical addresses or primitive values. This map symbolically represents the input memory graph at the beginning of an execution. The reason that CUTE introduces logical addresses is that actual concrete addresses of dynamically allocated cells may change in different executions. Also, the concrete addresses themselves are not necessary to represent memory graphs; it suffices to know how the cells are connected. Finally, CUTE attempts to make consecutive inputs similar, and this can be done with logical addresses. If CUTE used the actual physical addresses, it would depend on `malloc` and `free` (to return the same addresses) and more importantly, it would need to handle destructive updates of the input by the code under test: after CUTE generates one

input, the code changes it, and CUTE would need to know what changed to reconstruct the next input.

Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{V}$ be the set of all primitive values. Then, $\mathcal{I}\colon \mathbb{N} \rightarrow \mathbb{N} \cup \mathbb{V}$. The values in the domain and the range of $\mathcal{I}$ belonging to the set $\mathbb{N}$ represents the logical addresses. We also assume that each logical address $l \in \mathbb{N}$ has a type associated with it. A type can be `T *` (a pointer of type `T`) (where `T` can be primitive type or struct type) or $\mathtt{T}_p$ (a primitive type). The function *typeOf*(l) returns this type. Let the function *sizeOf*(`T`) returns the number of memory cells that an object of type `T` uses. If *typeOf*(l) is `T *` and $\mathcal{I}(l) \neq$`NULL`, then the sequence $\mathcal{I}(v), \ldots, \mathcal{I}(v+n-1)$ stores the value of the object pointed by the logical address l (each element in the sequence represents the content of each cell of the object in order), where $v = \mathcal{I}(l)$ and $n =$*sizeOf*(`T`). This representation of a logical input map essentially gives a simple way to serialize a memory graph.

We illustrate logical inputs on an example. Recall the example Input 3 from Figure 2.1. CUTE represents this input with the following logical input: $\langle 3, 1, 3, 0\rangle$, where logical addresses range from 1 to 4. The first value 3 corresponds to the value of `p`: it points to the location with logical address 3. The second value 1 corresponds to `x`. The third value corresponds to `p->v` and the fourth to `p->next` (0 represents `NULL`). This logical input encodes a set of concrete inputs that have the same underlying graph but reside at different concrete addresses. Similarly, the logical input map for Input 4 from Figure 2.1 is $\langle 3, 1, 3, 3\rangle$.

4.1.2 Instrumentation

To test a program P, CUTE tries to explore all execution paths of P. To explore all paths, CUTE first instruments the program under test. Then, it repeatedly runs the instrumented program P as follows:

// input: P is the instrumented program to test
// *depth* is the depth of bounded DFS
run_CUTE(P,*depth*)
  $\mathcal{I} = [\,]$; $h =$ (number of arguments in P) + 1;

Before Instrumentation	After Instrumentation
// program start l: `START`	*global vars* $\mathcal{A} = \mathcal{P} = path_c = M = [\,]$; *global vars* i =*inputNumber*= 0; l: `START`
// input l: $v \leftarrow$ *input()*;	l: *inputNumber* = *inputNumber*+1; *init_input*(&v, *inputNumber*);
// input l: $*v \leftarrow$ *input()*;	l: *inputNumber* = *inputNumber*+1; *init_input*(v, *inputNumber*);
// assignment l: $v \leftarrow e$;	l: *execute_symbolic*(&v,"e"); $v \leftarrow e$;
// assignment l: $*v \leftarrow e$;	l: *execute_symbolic*(v,"e"); $*v \leftarrow e$;
// conditional l: `if` (p) `goto` l	l: *evaluate_predicate*("p", p); `if` (p) `goto` l
// normal termination l: `HALT`	l: *compute_next_input*(); `HALT`;
// program error l: `ERROR`	l: **print** "Found Error with Input " . $\mathcal{I}$; *compute_next_input*(); `ERROR`;

Table 4.1: Code that CUTE's Instrumentation Adds

completed=**false**; *branch_hist*=[ ];

while not *completed*

execute P

Before starting the execution loop, CUTE initializes the logical input map $\mathcal{I}$ to an empty map and the variable h representing the next available logical address to the number of arguments to the instrumented program plus one. (CUTE gives a logical address to each argument at the very beginning.) The integer variable *depth* specifies the depth in the bounded DFS described in Section 4.1.4.

Table 4.1 shows the code that CUTE adds during instrumentation. The expressions enclosed in double quotes ("e") represent syntactic objects. We describe the instrumentation for function calls in Section 7.1. In the following section, we describe the various global variables and procedures that CUTE inserts.

4.1.3 Concolic Execution

Recall that a program instrumented by CUTE runs concretely and at the same time performs symbolic computation through the instrumented function calls. The symbolic execution follows the path taken by the concrete execution and replaces with the concrete value any symbolic expression that cannot be handled by our constraint solver.

An instrumented program maintains at the runtime two *symbolic states* $\mathcal{A}$ and $\mathcal{P}$, where $\mathcal{A}$ maps memory locations to *symbolic arithmetic expressions*, and $\mathcal{P}$ maps memory locations to *symbolic pointer expressions*. The symbolic arithmetic expressions in CUTE are linear, i.e. of the form $a_1x_1 + \ldots + a_nx_n + c$, where $n \geq 1$, each x_i is a symbolic variable, each a_i is an integer constant, and c is an integer constant. Note that n must be greater than 0. Otherwise, the expression is a constant, and CUTE does not keep constant expressions in $\mathcal{A}$, because it keeps $\mathcal{A}$ small: if a symbolic expression is constant, its value can be obtained from the concrete state. The arithmetic constraints are of the form $a_1x_1 + \ldots + a_nx_n + c \bowtie 0$, where $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$. The pointer expressions are simpler: each is of the form x_p, where x_p is a symbolic variable, or the constant `NULL`. The pointer constraints are of the form $x \cong y$ or $x \cong$ `NULL`, where $\cong \in \{=, \neq\}$.

Given any map $\mathcal{M}$ (e.g., $\mathcal{A}$ or $\mathcal{P}$), we use $\mathcal{M}' = \mathcal{M}[m \mapsto v]$ to denote the map that is the same as $\mathcal{M}$ except that $\mathcal{M}'(m) = v$. We use $\mathcal{M}' = \mathcal{M} - m$ to denote the map that is the same as $\mathcal{M}$ except that $\mathcal{M}'(m)$ is undefined. We say $m \in \mathit{domain}(\mathcal{M})$ if $\mathcal{M}(m)$ is defined.

Input Initialization using Logical Input Map

Figure 4.2 shows the procedure $\mathit{init_input}(m, l)$ that uses the logical input map $\mathcal{I}$ to initialize the memory location m, to update the symbolic states $\mathcal{A}$ and $\mathcal{P}$, and to update the input map $\mathcal{I}$ with new mappings.

M maps logical addresses to physical addresses of memory cells already allocated in an execution, and $\mathit{malloc}(n)$ allocates n fresh cells for an object of size n and returns the addresses of these cells as a sequence. The global variable h keeps track of the next unused logical address available for a newly allocated object.

For a logical address l passed as an argument to *init_input*, $\mathcal{I}(l)$ can be undefined in two cases: (1) in the first execution when $\mathcal{I}$ is the empty map, and (2) when l is some logical address that

```
// input: m is the physical address to initialize
//        l is the corresponding logical address
// modifies h, I, A, P
init_input(m, l)
    if l ∉ domain(I)
        if (typeOf(*m) ==pointer to T) *m =NULL;
        else *m =random();
        I = I[l ↦ *m];
    else
        v' = v = I(l);
        if (typeOf(v) ==pointer to T)
            if (v ∈ domain(M))
                *m = M(v);
            else
                n = sizeOf(T);
                {m_1, ..., m_n} =malloc(n);
                if (v ==non-NULL)
                    v' = h; h = h + n; // h is the next logical address
                *m = m_1; I = I[l ↦ v']; M = M[v ↦ m_1];
                for j = 1 to n
                    init_input(m_j, v' + j - 1);
        else
            *m = v; I = I[l ↦ v];
    // x_l is a symbolic variable for logical address l
    if (typeOf(m) ==pointer to T) P = P[m ↦ x_l];
    else A = A[m ↦ x_l];
```

Figure 4.2: Input Initialization

got allocated in the process of initialization. If $\mathcal{I}(l)$ is undefined and if *typeOf*(l) is not a pointer, then the content of the memory is initialized *randomly*; otherwise, if the *typeOf*(l) is a pointer, then the contents of l and m are both initialized to `NULL`. Note that CUTE does not attempt to generate random pointer graphs but assigns all new pointers to `NULL`. If *typeOf*$(\mathcal{I}(l))$ is a pointer to `T` (i.e., `T*`) and $M(l)$ is defined, then we know that the object pointed by the logical address l is already allocated and we simply initialize the content of m by $M(l)$. Otherwise, we allocate sufficient physical memory for the object pointed by `*m` using *malloc* and initialize them recursively. In the process, we also allocate logical addresses by incrementing h if necessary.

Symbolic Execution

Figure 4.3 shows the pseudo-code for the symbolic manipulations done by the procedure *execute_symbolic* which is inserted by CUTE in the program under test during instrumentation. The procedure *execute_symbolic*(m, e) evaluates the expression e symbolically and maps it to the memory location m in the appropriate symbolic state.

Recall that CUTE replaces a symbolic expression that the CUTE's constraint solver cannot handle with the concrete value from the execution. Assume, for instance, that the solver can solve only linear constraints. In particular, when a symbolic expression becomes non-linear, as in the multiplication of two non-constant sub-expressions, CUTE simplifies the symbolic expression by replacing one of the sub-expressions by its current concrete value (see line L in Figure. 4.3). Similarly, if the statement is for instance $v'' \leftarrow v/v'$ (see line D in Figure. 4.3), and both v and v' are symbolic, CUTE removes the memory location $\&v''$ from both $\mathcal{A}$ and $\mathcal{P}$ to reflect the fact that the symbolic value for v'' is undefined.

Figure 4.4 shows the function *evaluate_predicate*(p, b) that symbolically evaluates p and updates *path_c*. In case of pointers, CUTE only considers predicates of the form $x = y$, $x \neq y$, $x =$`NULL`, and $x \neq$`NULL`, where x and y are symbolic pointer variables. We discuss this in Section 4.1.6. If a symbolic predicate expression is constant, then `true` or `false` is returned.

At the time symbolic evaluation of predicates in the procedure *evaluate_predicate*, symbolic predicate expressions from branching points are collected in the array *path_c*. At the end of the execution, *path_c*$[0 \ldots i-1]$, where i is the number of conditional statements of P that CUTE executes, contains all predicates whose *conjunction* holds for the execution path.

Note that in both the procedures *execute_symbolic* and *evaluate_predicate*, we skip symbolic execution if the number of predicates executed so far (recorded in the global variable i) becomes greater than the parameter *depth* which gives the depth of bounded DFS described next.

4.1.4 Bounded Depth-First Search

To explore paths in the execution tree, CUTE implements a *(bounded) depth-first* strategy (bounded DFS). In the bounded DFS, each run (except the first) is executed with the help of a record of the conditional statements (which is the array *branch_hist*) executed in the previous run. For

```
// inputs: m is a memory location
//         e is an expression to evaluate
// modifies A and P by symbolically executing *m ← e
execute_symbolic(m, e)
    if (i ≤ depth)
        match e:
            case "v1":
                m1 = &v1;
                if (m1 ∈ domain(P))
                    A = A − m; P = P[m ↦ P(m1)]; // remove if A contains m
                else if (m1 ∈ domain(A))
                    A = A[m ↦ A(m1)]; P = P − m;
                else P = P − m; A = A − m;
            case "v1 ± v2": // where ± ∈ {+, −}
                m1 = &v1; m2 = &v2;
                if (m1 ∈ domain(A) and m2 ∈ domain(A))
                    v = "A(m1) ± A(m2)"; // symbolic addition or subtraction
                else if (m1 ∈ domain(A))
                    v = "A(m1) ± v2"; // symbolic addition or subtraction
                else if (m2 ∈ domain(A))
                    v = "v1 ± A(m2)"; // symbolic addition or subtraction
                else A = A − m; P = P − m; return;
                A = A[m ↦ v]; P = P − m;
            case "v1 * v2":
                m1 = &v1; m2 = &v2;
                if (m1 ∈ domain(A) and m2 ∈ domain(A))
    L:          v = "v1 * A(m2)"; // replace one with concrete value
                else if (m1 ∈ domain(A))
                    v = "A(m1) * v2"; // symbolic multiplication
                else if (m2 ∈ domain(A))
                    v = "v1 * A(m2)"; // symbolic multiplication
                else A = A − m; P = P − m; return;
                A = A[m ↦ v]; P = P − m;
            case "*v1":
                m2 = v1;
                if (m2 ∈ domain(P)) A = A − m; P = P[m ↦ P(m2)];
                else if (m2 ∈ domain(A)) A = A[m ↦ A(m2)]; P = P − m;
                     else A = A − m; P = P − m;
            default:
    D:      A = A − m; P = P − m;
```

Figure 4.3: Symbolic Execution

```
// inputs: p is a predicate to evaluate
//         b is the concrete value of the predicate in S
// modifies path_c, i
evaluate_predicate(p, b)
    if (i ≤ depth)
        match p:
            case "v1 ⋈ v2": // where ⋈ ∈ {<, ≤, ≥, >}
                m1 = &v1; m2 = &v2;
                if (m1 ∈ domain(A) and m2 ∈ domain(A))
                    c = "A(m1) − A(m2) ⋈ 0";
                else if (m1 ∈ domain(A))
                    c = "A(m1) − v2 ⋈ 0";
                else if (m2 ∈ domain(A))
                    c = "v1 − A(m2) ⋈ 0";
                else c = b;
            case "v1 ≅ v2": // where ≅ ∈ {=, ≠}
                m1 = &v1; m2 = &v2;
                if (m1 ∈ domain(P) and m2 ∈ domain(P))
                    c = "P(m1) ≅ P(m2)";
                else if (m1 ∈ domain(P) and v2 ==NULL)
                    c = "P(m1) ≅ NULL";
                else if (m2 ∈ domain(P) and v1 ==NULL)
                    c = "P(m2) ≅ NULL";
                else if (m1 ∈ domain(A) and m2 ∈ domain(A))
                    c = "A(m1) − A(m2) ≅ 0";
                else if (m1 ∈ domain(A)) c = "A(m1) − v2 ≅ 0";
                else if (m2 ∈ domain(A)) c = "v1 − A(m2) ≅ 0";
                else c = b;
        if (b) path_c[i] = c;
        else path_c[i] =neg(c);
    cmp_n_set_branch_hist(b);
    i = i + 1;
```

Figure 4.4: Symbolic Evaluation of Predicates

each conditional, CUTE records a *branch* value which is either `true` (if the *then* branch is taken) or `false` (if the *else* branch is taken), as well as a *done* value which is `false` when only one branch of the conditional has executed in prior runs (with the same history up to the branch point) and is `true` otherwise. The information associated with each conditional statement of the last execution path is stored in the array *branch_hist*, kept in a file between executions. For i, $0 \leq i < |branch_hist|$, $branch_hist[i] = (branch_hist[i].branch,\ branch_hist[i].done)$ is the record corresponding to the $(i+1)^{\text{th}}$ conditional executed.

The procedure *cmp_and_set_branch_hist* in Figure 4.5 checks whether the current execution path matches the one predicted at the end of the previous execution and represented in *branch_hist* passed between runs. Specifically, our algorithm maintains the invariant that when *run_program* is called, $branch_hist[|branch_hist| - 1].done$=`false` holds. This value is changed to `true` if the execution proceeds according to all the branches in *branch_hist* as checked by *cmp_n_set_branch_hist*. Note that we use the parameter *depth* to restrict the depth of search in the bounded DFS. We observed in our experiments that the execution almost always follows a prediction of the outcome of a conditional. However, it could happen that a prediction is not fulfilled because CUTE approximates, when necessary, symbolic expressions with concrete values (as explained in Section 4.1.3), and the constraint solver could then produce a solution that changes the outcome of some earlier branch. (Note that even when there is an approximation, the solution does not necessary change the outcome.) If it ever happens that a prediction is not fulfilled, an exception is raised to restart *run_CUTE* with a fresh random input.

Bounded depth-first search proves useful when the length of execution paths are infinite or long enough to prevent exhaustively search the whole computation tree. Particularly, it is important for generating finite sized data structures when using preconditions such as data structure invariants (see section 4.2. For example, if we use an invariant to generate sorted binary trees, then a non-bounded depth-first search would end up generating infinite number of trees whose every node has at most one left children and no right children.

```
// modifies branch_hist
cmp_n_set_branch_hist(branch)
    if (i < |branch_hist|)
        if (branch_hist[i].branch≠branch)
            print "Prediction Failed";
            raise an exception; // restart run_CUTE
        else if (i == |branch_hist| − 1)
            branch_hist[i].done =true;
    else branch_hist[i].branch = branch;
        branch_hist[i].done = false;
```

Figure 4.5: Prediction Checking

```
// modifies branch_hist, I, completed
compute_next_input()
    j = i − 1;
    while (j ≥ 0)
        if (branch_hist[j].done == false)
            branch_hist[j].branch= ¬branch_hist[j].branch;
            if (∃I′ that satisfies neg_last(path_c[0 . . . j]))
                branch_hist=branch_hist[0 . . . j];
                I = I′;
                return;
            else j = j − 1;
        else j = j − 1;
    if (j < 0) completed=true;
```

Figure 4.6: Compute Next Input

4.1.5 Computing an Input

After the termination of the instrumented program execution, an input for the next execution is computed using the procedure `compute_next_input` (see Figure 4.6). The procedure computes the input that will direct the next program execution along an alternative execution path. To do so, it loops over the elements of *path_c* from the end until it has generated an input that would force the program in the next execution to execute the last unexplored and feasible branch of a conditional along the current execution. Specifically, the procedure finds the last constraint *path_c*[*j*] such that the following holds:

- *path_c*[*j*] that has not been negated before, and

- there exists a logical input map $\mathcal{I}'$ such that $\mathcal{I}'$ is a satisfying solution of $neg_last(path_c[0 \ldots j])$, where $neg_last(path_c[0 \ldots j])$ denote the expression $path_c[0] \wedge \ldots \wedge path_c[j-1] \wedge \neg\ path_c[j]$, where only the last predicate is negated.

If such a constraint is found then $\mathcal{I}'$ is used as the input for the next execution. Otherwise, concolic testing is terminated indicating the completion of a depth-first search.

We next present how CUTE solves path constraints. Given a path constraint C=$neg_last(path_c[0 \ldots j])$, CUTE checks if C is satisfiable, and if so, finds a satisfying solution $\mathcal{I}'$.

We have implemented an *incremental* constraint solver for CUTE to optimize solving of the path constraints that arise in concolic execution. Our solver is built on top of `lp_solve` [61], a constraint solver for linear arithmetic constraints. Our solver provides three important optimizations for path constraints:

(OPT 1) Fast unsatisfiability check: The solver checks if the last constraint is syntactically the negation of any preceding constraint; if it is, the solver does not need to invoke the expensive semantic check. (Experimental results show that this optimization reduces the number of semantic checks by 60-95%.)

(OPT 2) Common sub-constraints elimination: The solver identifies and eliminates common arithmetic sub-constraints before passing them to the `lp_solve`. (This simple optimization, along with the next one, is significant in practice as it can reduce the number of sub-constraints by 64% to 90%.)

(OPT 3) Incremental solving: The solver identifies dependency between sub-constraints and exploits it to solve the constraints faster and keep the solutions similar. We explain this optimization in detail.

Given a predicate p in C, we define $vars(p)$ to be the set of all symbolic variables that appear in p. Given two predicates p and p' in C, we say that p and p' are *dependent* if one of the following conditions holds:

1. $vars(p) \cap vars(p') \neq \emptyset$, or
2. there exists a predicate p'' in C such that p and p'' are dependent and p' and p'' are dependent.

Two predicates are independent if they are not dependent.

The following is an important observation about the path constraints C and C' from two consecutive concolic executions: C and C' differ in the small number of predicates (more precisely, only in the last predicate when there is no backtracking), and thus their respective solutions $\mathcal{I}$ and $\mathcal{I}'$ must agree on many mappings. Our solver exploits this observation to provide more efficient, incremental constraint solving. The solver collects all the predicates in C that are dependent on $\neg path_c[j]$. Let this set of predicates be D. Note that all predicates in D are either linear arithmetic predicates or pointer predicates, because no predicate in C contains both arithmetic symbolic variables and pointer symbolic variables. The solver then finds a solution $\mathcal{I}''$ for the conjunction of all predicates from D. The input for the next run is then $\mathcal{I}' = \mathcal{I}[\mathcal{I}'']$ which is the same as $\mathcal{I}$ except that for every l for which $\mathcal{I}''(l)$ is defined, $\mathcal{I}'(l) = \mathcal{I}''(l)$. In practice, we have found that the size of D is almost *one-eighth* the size of C on average.

If all predicates in D are linear arithmetic predicates, then CUTE uses *integer linear programming* to compute $\mathcal{I}''$. If all predicates in D are pointer predicates, then CUTE uses the following procedure to compute $\mathcal{I}''$.

Let us consider only pointer constraints which are either equalities or disequalities. The solver first builds an equivalence graph based on (dis)equalities (similar to checking satisfiability in theory of equality [12]) and then based on this graph, assigns values to pointers. The values assigned to the pointers can be a logical address in the domain of $\mathcal{I}$, the constant `non-NULL` (a special constant), or the constant `NULL` (represented by 0). The solver views `NULL` as a symbolic variable. Thus, all predicates in D are of the form $x = y$ or $x \neq y$, where x and y are symbolic variables. Let D' be the subset of D that does not contain the predicate $\neg path_c[j]$. The solver first checks if $\neg path_c[j]$ is consistent with the predicates in D. For this, the solver constructs an undirected graph whose nodes are the equivalence classes (with respect to the relation $=$) of all symbolic variables that appear in D'. We use $[x]_=$ to denote the equivalence class of the symbolic variable x. Given two nodes denoted by the equivalence classes $[x]_=$ and $[y]_=$, the solver adds an edge between $[x]_=$ and $[y]_=$ iff there exists symbolic variables u and v such that $u \neq v$ exists in D' and $u \in [x]_=$ and $v \in [y]_=$. Given the graph, the solver finds that $\neg path_c[j]$ is satisfiable if $\neg path_c[j]$ is of the form $x = y$ and there is no edge between $[x]_=$ and $[y]_=$ in the graph; otherwise, if $\neg path_c[j]$ is of the form $x \neq y$, then $\neg path_c[j]$ is satisfiable if $[x]_=$ and $[y]_=$ are not the same equivalence class. If

```
// inputs: p is a symbolic pointer predicate
//         I is the previous solution
// returns: a new solution I''
solve_pointer(p, I)
    match p:
        case "x ≠NULL": I'' = {y ↦non-NULL| y ∈ [x]_=};
        case "x =NULL": I'' = {y ↦NULL| y ∈ [x]_=};
        case "x = y": I'' = {z ↦ v | z ∈ [y]_= and I(x) = v};
        case "x ≠ y": I'' = {z ↦ non-NULL| z ∈ [y]_=};
    return I'';
```

Figure 4.7: Assigning Values to Pointers

$\neg path_c[j]$ is satisfiable, the solver computes $\mathcal{I}''$ using the procedure *solve_pointer*$(\neg path_c[j], \mathcal{I})$ shown in Figure 4.7.

Note that after solving the pointer constraints, we either add (by assigning a pointer to `non-NULL`) or remove a node (by assigning a pointer `NULL`) from the current input graph, or alias or non-alias two existing pointers. This keeps the consecutive solutions similar. Keeping consecutive solutions for pointers similar is important because of the logical input map: if inputs were very different, CUTE would need to rebuild parts of the logical input map.

4.1.6 Approximations for Scalable Symbolic Execution

CUTE uses simple symbolic expressions for pointers and builds only (dis)equality constraints for pointers. We believe that these constraints, which approximate the exact path condition, are a good trade-off. To exactly track the pointer constraints, it would be necessary to use the theory of arrays/memory with updates and selections [66]. However, it would make the symbolic execution more expensive and could result in constraints whose solution is intractable. Therefore, CUTE does not use the theory of arrays but handles arrays by concretely instantiating them and making each element of the array a scalar symbolic variable.

It is important to note that, although CUTE uses simple pointer constraints, it still keeps a precise relationship between pointers: the logical input map (through types), maintains a relationship between pointers to structs and their fields and between pointers to arrays and their elements. For example, from the logical input map $\langle 3, 1, 3, 0\rangle$ for Input 3 from Figure 2.1, CUTE knows that

`p->next` is at the (logical) address 4 because `p` has value 3, and the field `next` is at the offset 1 in the `struct cell`. Indeed, the logical input map allows CUTE to use only simple scalar symbolic variables to represent the memory and still obtain fairly precise constraints.

Finally, we show that CUTE does not keep the exact pointer constraints. Consider for example the code snippet `*p=0; *q=1; if (*p == 1) ERROR` (and assume that `p` and `q` are not `NULL`). CUTE cannot generate the constraint `p==q` that would enable the program to take the "then" branch. This is because the program contains no conditional that can generate the constraint. Analogously, for the code snippet `a[i]=0; a[j]=1; if (a[i]==0) ERROR`, CUTE cannot generate `i==j`.

4.2 Data Structure Testing

We next consider testing of functions that take data structures as inputs. More precisely, a function has some pointer arguments, and the memory graph reachable from the pointers forms a data structure. For instance, consider testing of a function that takes a list and removes an element from it. We cannot simply test such function in isolation [115, 16, 118]—say generating random memory graphs as inputs—because the function requires the input memory graph to satisfy the data structure *invariant*.[2] If an input is invalid (i.e., violates the invariant), the function provides no guarantees and may even result in an error. For instance, a function that expects an acyclic list may loop infinitely given a cyclic list, whereas a function that expects a cyclic list may dereference `NULL` given an acyclic list. We want to test such functions with valid inputs only. There are two main approaches to obtaining valid inputs: (1) generating inputs with call sequences [115, 118] and (2) solving data structure invariants [16, 115]. CUTE supports both approaches.

4.2.1 Generating Inputs with Call Sequences

One approach to generating data structures is to use sequences of function calls. Each data structure implements functions for several basic operations such as creating an empty structure, adding an element to the structure, removing an element from the structure, and checking if an element is

[2]The functions may have additional preconditions, but we omit them for brevity of discussion; for more details, see [16].

in the structure. A sequence of these operations can be used to generate an instance of data structure, e.g., we can create an empty list and add several elements to it. This approach has two requirements [115]: (1) all functions must be available (and thus we cannot test each function in isolation), and (2) all functions must be used in generation: for complex data structures, e.g., red-black trees, there are memory graphs that cannot be constructed through additions only but require removals [115, 118].

4.2.2 Solving Data Structure Invariants

Another approach to generating data structures is to use the functions that check invariants. Good programming practice suggests that data structures provide such functions. For example, SGLIB [102] (see Section 7.2.2) is a popular C library for generic data structures that provides such functions. We call these functions `repOk` [16]. (SGLIB calls them `check_consistency`.) As an illustration, SGLIB implements operations on doubly linked lists and provides a `repOk` function that checks if a memory graph is a valid doubly linked list; each `repOk` function returns `true` or `false` to indicate the validity of the input graph.

The main idea of using `repOk` functions for testing is to *solve* `repOk` functions, i.e., generate only the input memory graphs for which `repOk` returns `true` [16, 115]. This approach allows modular testing of functions that implement data structure operations (i.e., does not require that all operations be available): all we need for a function under test is a corresponding `repOk` function. Previous techniques for solving `repOk` functions include a search that uses purely concrete execution [16] and a search that uses symbolic execution for primitive data but concrete values for pointers [115]. CUTE, in contrast, uses symbolic execution for both primitive data and pointers.

The constraints that CUTE builds and solves for pointers allow it to solve `repOk` functions asymptotically faster than the fastest previous techniques [16, 115]. Consider, for example, the following check from the invariant for doubly linked list: for each node `n`, `n.next.prev == n`. Assume that the solver is building a doubly linked list with N nodes reachable along the `next` pointers. Assume also that the solver needs to set the values for the `prev` pointers. Executing the check once, CUTE finds the exact value for each `prev` pointer and thus takes $O(N)$ steps to find the values for all N `prev` pointers. In contrast, the previous techniques [16, 115] take $O(N^2)$ steps

as they search for the value for each pointer, trying first the value `NULL`, then a pointer to the head of the list, then a pointer to the second element and so on.

4.3 Discussion

We next discuss the advantages of using CUTE over traditional symbolic execution based testing approaches.

4.3.1 Pointer Casting and Arithmetic

CUTE often has an advantage over static analysis in reasoning about linked data. For example, to determine if two pointers point to the same memory location, CUTE simply checks whether their values are equal and does not require an alias analysis that may be inaccurate in the presence of pointer casting and pointer arithmetic. For example, for the following C program:

```
struct foo {
  int i;
  char c; };

void * memset(void *s,char c,size_t n) {
  for (int i = 0; i < n; i++)
     ((char *)s)[i] = c; return s;
}

bar (struct foo *a) {
  if (a && a->c == 1) {
    memset(a,0,sizeof(struct foo));
    if (a->c != 1)
      ERROR;
  }
}
```

a fully sound static analysis should report that `ERROR` might be reachable. However, such a sound static-analysis tool would be impractical as it would give too many false alarms. More practical tools, such as BLAST [51], report that the code is safe because a standard alias analysis is not able to see that `a->c` has been overwritten. In contrast, CUTE easily finds a precise execution leading to the `ERROR`. This kind of code is often found in C where `memset` is widely used for fast memory initialization.

4.3.2 Library Functions with Side-Effects

The concrete execution of CUTE helps to remove false alarms, especially in the presence of library function calls that can have side-effects. In the above code, for example, if the function `memset` is a library function with no source code available, static-analysis tools have no way to find out how the function can affect the global heap. In such situations, they definitely give false alarms. However, CUTE can tackle the situation as it can see the side-effect while executing the function concretely.

4.3.3 Approximating Symbolic Values by Concrete Values

CUTE combines the concrete and symbolic executions to make them co-operate with each other, which helps to handle situations where most symbolic executors would give uncertain results. For example, consider testing the function `f` in the following C code:

```
g(int x) {
   return x * x + (x % 2);
}

f(int x, int y) {
   z = g(x);
   if (y == z)
      ERROR;
}
```

A symbolic executor would generate the path constraint `y = x * x + (x % 2)` that is not in a decidable theory. Thus, it cannot say that `ERROR` is reachable with guarantee. On the other hand, suppose that CUTE starts with the initial inputs `x = 3, y = 4`. In the first execution, since CUTE cannot handle the symbolic expression `x * x + (x % 2)`, it approximates `z` by the concrete value `3 * 3 + (3 % 2) = 10` and proceeds to generate the path constraint `y != 10`. Therefore, by solving the path constraint CUTE will generate the inputs `x = 3, y = 10` for the next run which will reveal the `ERROR`.

4.3.4 Black-Box Library Functions

The same situation arises in the above code if `g` is a library function whose source code is not available. A symbolic executor would generate the path constraint `y = g(x)` involving uninterpreted function and would give a possible warning. However, CUTE in the same way as before generates an input leading to the `ERROR`.

4.3.5 Lazy Initialization

One can imagine combining symbolic execution with randomization in several ways. CUTE commits to concrete values before the execution. Another approach would be to use full symbolic execution and generate concrete values on demand [115]. However, this approach does not handle black-box library functions, executes slower as it needs to always check if data is initialized, and cannot "recover" from bad initialization as this example shows:

```
f(int x){
   z = x * x + (x % 2);
   if (z == 8) {
      ERROR;
   }
   if (x == 10) {
      ERROR;
   }
}
```

After executing the first `if` statement, a lazy initializer will initialize `x` to a random value in any run since it cannot decide the path constraint `x * x + (x % 2) = 8`. Thus, it would not be able to take the `then` branch of the second `if`. CUTE, however, would generate `x = 10` for the second run as it simultaneously executes both concretely and symbolically.

4.3.6 Random Initialization

Concolic testing uses random values to initialize the input values. This helps concolic testing to mitigate the limitations of the constraint solver in many situations. To illustrate this, consider the following C program:

```
f(int x, int y){
  if (x*x*x > 0){
     if (x>0 && y==10)
        ERROR;
  } else {
     if (x>0 && y==20)
        ERROR;
  }
}
```

Given a theorem prover that cannot reason about non-linear arithmetic constraints, a static analysis tool using predicate abstraction [11, 51] will report that both ERRORs in the above code may be reachable; therefore, the tool will give one false alarm since the second ERROR is unreachable. This would be true as well if the test `(x*x*x > 0)` is replaced by a black-box library call. On the other hand, a test-generation tool based on symbolic execution [115] will not be able to generate an input to detect any ERROR because its symbolic execution will be stuck at the condition of the first if-then-else statement. In contrast, CUTE can randomly generate an input where `x>0` and `y!=10` with almost 0.5 probability; after the first execution with such an input, the depth-first search of CUTE will generate another input with the same positive value of `x` but with `y=10`, which will lead the program in its second run to the first ERROR. Note that if CUTE randomly generates a negative value for `x` in the first run, then CUTE will generate an input where `x>0` and `y==20`

to satisfy the then branch of the third if-then-else statement (it will do so because no constraint is generated for the condition of the first if-then-else statement since it is non-linear); however, due to the concrete execution, CUTE will then not take the `else` branch of the first if-then-else statement in such a second run. In summary, our mixed strategy of random and directed search along with simultaneous concrete and symbolic execution of the program will allow us to find the only reachable ERROR statement in the above example with high probability.

Chapter 5

Testing Concurrent Programs

In this chapter, we extend concolic testing with a new algorithm, called *the race-detection and flipping algorithm*,[1] to effectively test shared-memory multi-threaded programs.

We described concolic testing to systematically and automatically test sequential programs. However, most of the real-world programs are concurrent where computation tasks are distributed among several threads executing simultaneously and communicating either through shared memory or message passing. Testing concurrent programs is notoriously harder than testing sequential programs because of the exponentially large number of possible interleavings of concurrent events [104]. Many of these interleavings share the same causal structure (also called the *partial order*), and are thus equivalent with respect to finding bugs in a given program. Techniques for avoiding such redundant executions are called *partial order reduction* [110, 76, 38].

Several approaches [39, 34, 19, 17] to testing concurrent programs assume that the data inputs are from a small finite domain. These approaches rely on exhaustively executing the program for all possible inputs and perform a partial order reduction to reduce the search space. The problem with these approaches is that it is hard to scale them—the input set is often too large.

A second approach is to execute a program symbolically in a customized virtual machine which supports partial order reduction [53, 115]. This requires checking satisfiability of complex constraints (corresponding to every branch point in a program). Unfortunately, checking such satisfiability may be undecidable or computationally intractable. Moreover, in concurrent programs, partial order reduction for symbolic execution requires computing the dependency relations between memory accesses in a program. Because it involves alias analysis, such a computation is often conservative resulting in extra dependencies. Therefore, large numbers of unreachable branches may be explored, often causing many warnings for bugs that could never occur in an actual execution.

[1] This work is partly done in collaboration with Gul Agha. Part of this work appeared in [86, 85].

Our approach is to extend *concolic testing* with a new technique called race-detection and flipping. To use concolic testing for multi-threaded programs, we do the following. For a given concrete execution, at runtime, we determine the causality relation or the *exact* race conditions (both data race and lock race) between the various events in the execution path. Subsequently, we systematically re-order or permute the events involved in these races by generating new thread schedules as well as generate new test inputs. This way we explore at least one representative from each partial order. The result is an efficient testing algorithm for concurrent programs which, at the cost of missing some potential bugs, avoids the problem of false warnings.

We have implemented the algorithm in a publicly available tool, called jCUTE, for testing Java programs. Apart from detecting assertion violations and uncaught exceptions, jCUTE reports all data race conditions and deadlock states encountered during the process of testing.

The rest of the chapter is organized as follows. In Section 5.1, we describe the race-detection and flipping algorithm on programs having no data inputs and prove the correctness of the algorithm. In Section 5.2, we describe a simple algorithm for extending concolic testing to shared-memory multi-threaded programs. The goal of this section is to familiarize the readers with various data structures used in the algorithms described in the subsequent sections. In Section 5.3, we show how to combine concolic testing with the race-detection and flipping algorithm. We propose a further optimization of the combined algorithm in Section 5.4. Finally, we conclude the chapter by discussing some of the advantages of the techniques described in this chapter.

5.1 The Race-Detection and Flipping Algorithm

In the description of the race-detection and flipping algorithm, we assume that a program under test has no data input. We make this simplification to keep the exposition concise and to keep the proof of correctness of the race-detection and flipping algorithm simple. In the subsequent sections, we show how we can combine the race-detection and flipping algorithm with concolic testing.

The race-detection and flipping algorithm is given in Figure 5.1. Recall that $\texttt{Ex}(P)$ is the set of all feasible execution paths that can be exhibited by the program P on all possible values of inputs and all possible schedules (see Section 3.2). Similarly, $\texttt{REx}(P)$ is the set that contains exactly one candidate from each equivalence class of feasible execution paths of P. *test_program*(P) repeatedly

```
global var τ = ϵ; // the empty sequence

//input: P is the program to test
test_program(P)
    while testing not completed
        execute_program(P)

execute_program(P)
    execute_prefix(P, τ);
    while there is an enabled thread
        execute the next statement of the lowest indexed enabled thread in P
            to generate the event e;
        race(τ) = false;
        postponed(τ) = ∅;
        append e to τ;
        if ∃e' ∈ τ such that e' ⋖ e
            let τ = τ1 e' τ2 in race(τ1) = true;
    // end of the while loop
    if there is an active thread
        print ''Error:  found deadlock'';
    generate_next_schedule();

// modifies τ
generate_next_schedule()
    if ∃e such that τ == τ1 e τ2 and backtrackable(τ1) and
        there is no e' such that τ == τ1' e' τ2' and |τ1| < |τ1'| and backtrackable(τ1')
            race(τ1) = false;
            let (t, _, _) = e in add t to postponed(τ1);
            let t = smallest indexed thread in enabled(τ1)\ postponed(τ1) in τ = τ1(t, _, _);
    else
        testing completed;

backtrackable(τ1) =
    race(τ1) ==true and |enabled(τ1)\postponed(τ1)| > 1
```

Figure 5.1: The Race-Detection and Flipping Algorithm

executes the program P with different schedules until all paths in a $\texttt{REx}(P)$ have been explored. Given two sequences of events τ and τ', we let $\tau\tau'$ to denote the concatenation of the two sequences. Similarly, given a sequence of events τ and an event e, we let τe to denote the concatenation of the sequence and the event. Let ϵ be the empty sequence. A sequence of events is called a *prefix*, if it is the prefix of a feasible execution path. The global variable τ keeps track of the execution path for each execution of P. At the end of each execution, τ is appropriately truncated so that a depth-first search of the computation tree takes place. *execute_prefix*(P, τ) executes the program from the beginning until the sequence of events generated by the execution is equal to the prefix τ. Since an execution path is solely determined by the sequence of threads that are executed in the path, from now onwards we will ignore the second and the third components of a tuple representing an event. Thus $(t, _, _)$ represents an event on the thread t. With every prefix τ, we associate a set, denoted by *postponed*(τ). Moreover, with every prefix τ, we associate a boolean flag, denoted by *race*(τ). *enabled*(τ) returns the set of threads that are enabled after executing the prefix τ. *enabled*$(\tau)\backslash$*postponed*(τ) represents the set of threads that are enabled but not postponed after executing τ.

In each execution of P during the testing process, P is first partly executed so that it follows the prefix τ computed in the previous execution. Then P is executed with the default schedule, where the lowest indexed enabled thread is always chosen. If $\tau = \tau' e$ before the start of an execution, then the execution path and the previous execution path has the same prefix τ'. In an execution path τ, for any prefix τ' of τ, we set *race*(τ') to **true**, if there exist e, τ_1, e', and τ_2 such that $\tau = \tau' e \tau_1 e' \tau_2$ and $e \lessdot e'$. Setting *race*(τ') to **true** flags that in a subsequent execution, we must postpone the execution of e after the prefix τ' so that we may explore a possibly non-equivalent execution path. At the end of an execution, if τ_1 is the longest prefix of the current execution path τ such that *race*(τ_1) is set to **true** and $|$*enabled*$(\tau_1)\backslash$*postponed*$(\tau_1)| > 1$, we generate a new schedule by truncating τ to $\tau_1 e$, where e is an event of a thread t that has not been scheduled after τ_1 in any previous execution.

We next prove that the race-detection and flipping algorithm explores all execution paths in a set $\texttt{REx}(P)$. To keep the proof simple, we assume that no execution path in P ends in a deadlock state.

Theorem 9. *If $\mathtt{Ex}'(P)$ is the set of the execution paths that are explored by the race-detection and flipping algorithm, then there is a set $\mathtt{REx}(P)$ such that $\mathtt{REx}(P) \subseteq \mathtt{Ex}'(P) \subseteq \mathtt{Ex}(P)$.*

Proof. Before we prove the theorem, we state and prove the following three lemmas.

Lemma 10. *If $e \lessdot e'$ in an execution path τ, then $e \lessdot e'$ in any execution path $\tau' \in [\tau]_{\equiv_\preccurlyeq}$*

The proof of the above lemma follows from the definition of $\preccurlyeq$.

Lemma 11. *Given an event e and a prefix τ, if the following conditions hold:*

1. *e is enabled after the prefix τ,*
2. *$\tau\tau' e\tau''$ is a feasible execution path,*
3. *there is no event e_1 in τ' such that $e_1 \lessdot e'$, and*
4. *there is an event e_2 in τ'' such that $e' \lessdot e_2$,*

then $\tau e\tau'\tau'' \equiv_\preccurlyeq \tau\tau' e\tau''$ and $e \lessdot e_2$ in $\tau e\tau'\tau''$.

Proof. $\tau e\tau'\tau'' \equiv_\preccurlyeq \tau\tau' e\tau''$ holds if there is no event e' in τ' such that $e' \preccurlyeq e$. Let us assume that there is an event e' in τ' such that $e' \preccurlyeq e$. This is possible in three cases.

i) There is an e_3 in τ' such that $e' \preccurlyeq e_3$ and $e_3 \lhd e$. Because e is enabled after both τ and $\tau\tau'$, for any e_4 in τ, $e_4 \Updownarrow e$. Therefore, there cannot be such an e_3.

ii) e is a read or a write event and there is an e_3 in τ' such that $e' \preccurlyeq e_3$ and e_3 be the latest event such that $e_3 <_m e$. Then $e_3 \lessdot e$. This contradicts condition (3).

iii) e is a lock acquire event, and there is an e_3 in τ' such that $e \preccurlyeq e_3$ and e_3 is the latest event such that $e_3 <_m e$. Let e_4 be the lock acquire event corresponding to the lock release event e_3. If e_4 is in τ, then e cannot be enabled immediately after τ. Hence, we have a contradiction. If e_4 is in τ', then $e_4 \lessdot e$. This contradicts condition (3).

Therefore, all the above three cases are impossible. This proves that $\tau e\tau'\tau'' \equiv_\preccurlyeq \tau\tau' e\tau''$.

$e \lessdot e_2$ follows from Lemma 10. □

Lemma 12. *If $\tau e \tau_1 e' \tau_2$ is a feasible execution path and if $e \lessdot e'$, then e' is enabled immediately after τ.*

Proof. Let e be generated by the thread t and e' be generated by the thread t'. Let L be the set of locks held by t immediately before e is generated and L' be the set of locks held by t' immediately before e' is generated. Then $L \cap L' = \emptyset$. If this is not the case, then let $l \in L \cap L'$. There exist two events e_1 and e_2 in τ_1, such that e_1 is an event of t, e_2 is an event of t', e_1 is the release of l, and e_2 is the acquire of l. Therefore, $e_1 \preccurlyeq e_2$. Moreover, $e \preccurlyeq e_1$ and $e_2 \preccurlyeq e'$. This violates the fact that $e \lessdot e'$. Therefore, $L \cap L' = \emptyset$. Moreover, $e \lessdot e'$ implies that $e \Updownarrow e'$. Hence, e' is enabled immediately after τ. □

Lemma 13. *If $\tau e \tau_1 e' \tau_2$ is a feasible execution path and if $e \lessdot e'$, then there exists a feasible execution path of the form $\tau e' \tau_3 e \tau_4$ such that $e' \lessdot e$ holds.*

Proof. By Lemma 12, e' is enabled after τ. Therefore, $\tau e'$ is a prefix. Because we assume that a path never ends in a deadlock state, there must exist a feasible execution path whose prefix is $\tau e'$ and e is in the path. Let $\tau e' \tau_3 e \tau_4$ be the feasible execution path in which after executing $\tau e'$, we execute e as soon as it gets enabled. In this path $e' \lessdot e$. □

We now get back to the proof of Theorem 9. The proof of $\texttt{Ex}'(P) \subseteq \texttt{Ex}(P)$ is straightforward. If we remove the check *race*(τ_1) == **true** from the function *backtrackable*(τ_1), then after each prefix τ_1, we explore the events of all enabled threads. Therefore, the algorithm explores the entire computation tree, that is, the algorithm explores all execution paths in $\texttt{Ex}(P)$. Clearly, if we keep the check *race*(τ_1) == **true** in the function *backtrackable*(τ_1), we may explore a smaller number of execution paths. Therefore, $\texttt{Ex}'(P) \subseteq \texttt{Ex}(P)$.

We next prove $\texttt{REx}(P) \subseteq \texttt{Ex}'(P)$ by induction. Specifically, we prove that for any $\tau_1 \in \texttt{Ex}(P)$, there is a $\tau_2 \in \texttt{Ex}'(P)$ such that $\tau_1 \equiv_{\preccurlyeq} \tau_2$. Let τ be a prefix explored by the algorithm. Let us define $\texttt{Ex}_\tau(P) = \{\tau' \mid \tau' \in \texttt{Ex}(P) \text{ and } \tau \text{ is a prefix of } \tau'\}$. Similarly, let $\texttt{Ex}'_\tau(P) = \{\tau' \mid \tau' \in \texttt{Ex}'(P) \text{ and } \tau \text{ is a prefix of } \tau'\}$. By an argument similar to that in the previous paragraph, $\texttt{Ex}'_\tau(P) \subseteq \texttt{Ex}_\tau(P)$.

Let $T_{en} = \{t_{i_1}, \ldots, t_{i_l}\}$ be the set of threads that are enabled after executing τ, where $i_p < i_q$ if and only if $p < q$. Let e_k be the event generated if we execute the thread t_{i_k} after τ. If we

remove the check $race(\tau_1) == \textbf{true}$ from the function $backtrackable(\tau_1)$, then after executing τ our algorithm will execute all the threads in T_{en} in various executions. However, if we keep the check $race(\tau_1) = \textbf{true}$ in the function $backtrackable(\tau_1)$, then after executing τ our algorithm may only execute the threads $t_{i_1}, t_{i_2}, \ldots, t_{i_k}$ in that order where $k < l$. This happens if after exploring all paths in $\texttt{Ex}'_{\tau e_k}(P)$, $race(\tau)$ is **false**. This happens if there exists no e in any of the paths in $\texttt{Ex}'_{\tau e_k}(P)$ such that $e_k \lessdot e$. By the induction hypothesis, let us assume that for each $\tau' \in \texttt{Ex}_{\tau e_k}(P)$, there is a $\tau'' \in \texttt{Ex}'_{\tau e_k}(P)$ such that $\tau' \equiv_{\preccurlyeq} \tau''$. We want to prove that the same holds for τ, that is, for each $\tau' \in \texttt{Ex}_{\tau}(P)$, there is a $\tau'' \in \texttt{Ex}'_{\tau}(P)$ such that $\tau' \equiv_{\preccurlyeq} \tau''$. This holds if we can show that for any $k+1 \leq m \leq l$ and any $\tau_1 \in \texttt{Ex}_{\tau e_m}(P)$, there is a $\tau_2 \in \texttt{Ex}'_{\tau e_k}(P)$ such that $\tau_1 \equiv_{\preccurlyeq} \tau_2$.

Consider $\tau_1 \in \texttt{Ex}_{\tau e_m}(P)$. Then τ_1 must be of the form $\tau e_m \tau' e_k \tau''$. We consider three cases as follows.

(i) If there exists no e in $e_m\tau'$ such that $e \lessdot e_k$, and there exists no $e' \in \tau''$ such that $e_k \lessdot e''$, then $\tau e_k e_m \tau' \tau'' \equiv_{\preccurlyeq} \tau e_m \tau' e_k \tau''$ by Lemma 11. But $\tau e_k e_m \tau' \tau'' \in \texttt{Ex}_{\tau e_k}(P)$. Therefore, by the induction hypothesis there exists $\tau_2 \in \texttt{Ex}'_{\tau e_k}(P)$ such that $\tau e_k e_m \tau' \tau'' \equiv_{\preccurlyeq} \tau_2$. This implies that $\tau_2 \equiv_{\preccurlyeq} \tau e_m \tau' e_k \tau''$. Therefore, the induction claim holds in this case.

(ii) Now we show the impossibility of the case that there exists no e in $e_m\tau'$ such that $e \lessdot e_k$, and there exists an e' in τ'' such that $e_k \lessdot e'$. Then $\tau e_k e_m \tau' \tau'' \equiv_{\preccurlyeq} \tau e_m \tau' e_k \tau''$ Lemma 11. Note that in $\tau e_k e_m \tau' \tau''$, the fact $e_k \lessdot e'$ still holds. By the induction hypothesis, there exists $\tau e_k \tau_4 e' \tau_5 \in \texttt{Ex}'_{\tau e_k}(P)$ such that $\tau e_k \tau_4 e' \tau_5 \equiv_{\preccurlyeq} \tau e_k e_m \tau' \tau''$. By Lemma 10, in $\tau e_k \tau_4 e' \tau_5$, the fact $e_k \lessdot e'$ holds. This contradicts the assumption that after executing t_{i_k} after τ, $race(\tau)$ is **false**.

(iii) Now we show the impossibility of the case that there exists an e in $e_m\tau'$ such that $e \lessdot e_k$. Let us assume that there exists an e in $e_m\tau'$ such that $e \lessdot e_k$. Then we show that this violates the assumption that after executing t_{i_k} after τ, $race(\tau)$ is **false**. Consider the following two cases.

(a) The first case is $e = e_m$. Since $e_m \lessdot e_k$ and $\tau e_m \tau' e_k \tau''$ is a feasible execution path of P, by Lemma 13, there is a feasible execution path of P of the form $\tau e_k \tau_4 e_m \tau_5$ such that in $\tau e_k \tau_4 e_m \tau_5$, the fact $e_k \lessdot e_m$ holds. This implies that $\tau e_k \tau_4 e_m \tau_5 \in \texttt{Ex}_{\tau e_k}(P)$. By

the induction hypothesis, there exists $\tau e_k \tau_6 e_m \tau_7 \in \mathtt{Ex}'_{\tau e_k}(P)$ such that $\tau e_k \tau_6 e_m \tau_7 \equiv_{\preccurlyeq} \tau e_k \tau_4 e_m \tau_5$. By Lemma 10, in $\tau e_k \tau_6 e_m \tau_7$, the fact $e_k \lessdot e_m$ holds. This violates the assumption that after executing t_{i_k} after τ, $race(\tau)$ is **false**.

(b) The second case is that e is present in τ'. Let $\tau_1 = \tau e_m \tau_8 e \tau_9 e_k \tau''$. Since $e \lessdot e_k$ and τ_1 is a feasible execution path of P, by Lemma 13, there is feasible execution path of P of the form $\tau e_m \tau_8 e_k \tau_4 e \tau_5$ such that in $\tau e_m \tau_8 e_k \tau_4 e \tau_5$, the fact $e_k \lessdot e_m$ holds. Therefore, $\tau e_m \tau_8 e_k \tau_4 e \tau_5 \in \mathtt{Ex}_{\tau e_m}(P)$. If there is no e' in $e_m \tau_8$ such that $e' \lessdot e_k$, then by case (ii), we have a contradiction. Otherwise, if there is a e' in $e_m \tau_8$ such that $e' \lessdot e_k$, then we get the case (iii) with $|\tau_8| < |\tau'|$. We repeat the process until we get the case (iii)(a) or the case (ii). The process is guaranteed to terminate as at each step the length of τ_8 gets smaller than the length of τ'. On the termination of the process we get a contradiction.

□

5.2 Extending Concolic Testing to Test Concurrent Programs

We next describe a simple algorithm for testing shared-memory multi-threaded programs with data inputs. The algorithm naïvely extends concolic testing: the algorithm does not combine concolic testing with the race-detection and flipping algorithm. Our goal in describing the simple algorithm is to familiarize the readers with the various data structures that we consistently use in our algorithms.

Given a program P in Scil, our simple algorithm explores a subset of the execution paths in $\mathtt{Ex}(P)$. This is done by repeatedly executing P both symbolically and concretely on different inputs and schedules, each of which leads the program along a different execution path. At the end of each execution of P, our algorithm either computes a new schedule or a new input, which is used in the next execution of P. A new input is generated by solving constraints. To generate a new schedule, our algorithm picks a scheduler choice recorded during the execution and generates a new schedule where the particular thread chosen in the scheduler choice is *postponed*.

5.2.1 Instrumentation

jCUTE first instruments the program P under test. Table 5.1 shows the code that jCUTE adds during instrumentation. The code added by jCUTE during instrumentation includes the code added by CUTE as in concolic testing. In addition, jCUTE also adds code so that it can control the interleaving of the various threads at runtime. Specifically, jCUTE adds a call to the procedures *access_event* before any access to a potential shared memory, *fork_event* after forking a new thread, and *end_event* after terminating a thread. After the `START` statement in a program, jCUTE adds code to create a new thread and execute the procedure *testing_scheduler* in the newly created thread. This procedure controls the execution of the other threads at runtime. As such jCUTE can execute a program according to a pre-determined schedule.

After instrumentation, jCUTE repeatedly runs the instrumented program P as follows:

```
// input: P is the instrumented program to test
//        depth is the depth of bounded DFS
run_jCUTE(P,depth)
    I = [ ]; h = (number of arguments in P) + 1;
    completed=false; branch_hist=[ ]; event=[ ]; postponed=[ ];
    while not completed
        execute P
```

In an execution, the global shared variable i counts the number of shared memory accesses and the number of conditional statement executions; i is incremented by 1 before any shared memory access or before the execution of any conditional statement. The execution points at which i is incremented denote *choice points*—the execution path of the program P can be changed at these choice points either by generating a different schedule or a different input. At these choice points we record the information required for generating new input in the arrays *branch_hist* and *path_c* and in the maps $\mathcal{I}$, $\mathcal{A}$, $\mathcal{P}$, and M. The use of these data structures is already described in Chapter 4. In addition, we record information required for generating new schedules in the arrays *event*, *enabled*, *postponed*, *race*. Since a choice point i can represent either a scheduler choice or

Before Instrumentation	After Instrumentation

// program start
l: `START`
	shared global vars $\mathcal{A} = \mathcal{P} = path_c = enabled = M = [\,]$;
	shared global vars $i = inputNumber = 0$;
	global var t_{current} = `NULL`; // stores the scheduled thread
	shared global var $race = [\,]$; // used in the efficient algorithm
	$sleep = delayed = \{\,\}$; // used in the optimized algorithm
	l: `START`
	create a new thread and execute *testing_scheduler*() in that thread

// input l: $lv \leftarrow input()$;	l: $inputNumber = inputNumber+1$; $init_input(\&lv, inputNumber)$;
// new thread l: $fork(l)$;	l: $fork(l)$; $fork_event()$;
// lock l: $lock(\&v)$;	l: $access_event(\&v, l, \mathbf{l})$; $lock(\&v)$;
// unlock l: $unlock(\&v)$;	l: $unlock(\&v)$; $access_event(\&v, l, \mathbf{u})$;
// assignment l: $v \leftarrow lv$;	l: $access_event(\&v, l, \mathbf{w})$; $execute_symbolic(\&v, \text{“}lv\text{”})$; $v \leftarrow lv$;
// assignment l: $*lv_1 \leftarrow lv_2$;	l: $access_event(lv_1, l, \mathbf{w})$; $execute_symbolic(lv_1, \text{“}lv_2\text{”})$; $*lv_1 \leftarrow lv_2$;
// assignment l: $lv \leftarrow v$;	l: $access_event(\&v, l, \mathbf{r})$; $execute_symbolic(\&lv, \text{“}v\text{”})$; $lv \leftarrow v$;
// assignment l: $lv \leftarrow *lv$;	l: $access_event(lv, l, \mathbf{r})$; $execute_symbolic(\&lv, \text{“}*lv\text{”})$; $lv \leftarrow *lv$;
// assignment l: $lv \leftarrow e$; where $e \in \{\&v, c, lv, lv\ op\ lv\}$	l: $execute_symbolic(\&v, \text{“}e\text{”})$; $lv \leftarrow e$;
// conditional l: `if` (p) `goto` l	l: $evaluate_predicate(\text{“}p\text{”}, p)$; `if` (p) `goto` l
// normal termination l: `HALT`	l: `HALT`; *end_event()*;
// program error l: `ERROR`	l: **print** “Found Error with Input ” . $\mathcal{I}$; `ERROR`; *end_event()*;

Table 5.1: Code that jCUTE's Instrumentation Adds

a branch choice, either the elements $event[i]$, $enabled[i]$, $postponed[i]$, $race[i]$ are defined or the elements $branch_hist[i]$, $path_c[i]$ are defined.

The array *event* is used to keep track of the sequence of events generated by an execution of P. Thus the array *event* serves the same purpose as the global variable τ in Figure 5.1. $postponed[i]$, if defined, contains a set of threads that cannot be executed in the next execution of P at the choice point i. This array, like the *event* array, maintains information across executions. The other two arrays *enabled* and *race* are not used across executions. $enabled[i]$, if defined, contains a set of threads that are not enabled at the choice point i. $race[i]$ is set to true if the event stored in $event[i]$ has an immediate race with some other future event in the execution path. The simple algorithm does not use this field; the field will be used in the efficient algorithm described in Section 5.3. If we use $event[0 \ldots i]$ to represent the sequence of events $event[0]event[1] \ldots event[i]$, then $postponed[i]$ is the same as $postponed(event[0 \ldots i-1])$ defined in Section 5.1. Similarly, $race[i]$ is the same as $race(event[0 \ldots i-1])$ defined in Section 5.1.

The global variable t_{current} is used by the procedure *testing_scheduler* to store the currently scheduled thread. The global sets *sleep* and *delayed* are used by the further optimized algorithm described in Section 5.4.

5.2.2 Controlling the Execution of Threads

At runtime, the execution of various threads is controlled by the thread executing the procedure *testing_scheduler*. This procedure, besides controlling the execution of various threads, ensures that at any time only one thread is executing. This serialization of the execution of various threads ensures that there is no uncontrolled concurrency in the system. We next describe how the procedure *testing_scheduler* controls the various threads so that it can systematically explore the feasible interleavings.

Let us denote the thread running the procedure *testing_scheduler* by *schedulerThread*. We use the variable *thisThread* to denote the current thread (i.e. the thread accessing the variable *thisThread*.) The execution of various threads is controlled using binary semaphores. The pseudo-code of the implementation of a binary semaphore is given in Figure 5.2. A call to the procedure *wait* on a semaphore s makes the calling thread wait until the value of s is 1. Once the value is

```
// Semaphore s is passed by reference in the following procedures
init(Semaphore s){
    s = 0 ;
}

// Also known as P()
wait(Semaphore s) {
    await s == 1, then s = 0 ; // must be atomic once s == 1 is detected
}

// Also known as V()
signal(Semaphore s) {
    s = 1 ; // must be atomic
}
```

Figure 5.2: Binary Semaphore

1, it sets the value of s to 0 atomically. A call to the procedure *signal* on a semaphore s sets the value of s to 1 atomically; this signals any thread waiting on s.

We associate a binary semaphore with each thread at the time of its creation and initialize it to 0. We use $t.semaphore$ to denote the semaphore associated with the thread t.

The definition of the various thread controlling procedures introduced through instrumentation is given in Figure 5.3. In an execution, before any access to a shared memory location, a thread, say t, calls the procedure *access_event*. This procedure first executes *signal*(*schedulerThread*. *semaphore*) to signal the *schedulerThread* thread to continue its execution. Then the procedure executes *wait*(*thisThread*. *semaphore*) to make t wait for a signal from the *schedulerThread* thread. This way t releases the control to the *schedulerThread* thread and allows *schedulerThread* to schedule an appropriate thread from the set of enabled threads. Note that, although, the thread trying to access the shared memory location is waiting on its semaphore, it is *enabled* by definition.

A thread also starts waiting on its semaphore when it forks another thread. However, in this case, the thread calling fork does not signal the thread *schedulerThread*. This is because after the execution of fork the child thread starts its execution and we want that at any time during an execution only one thread is executing.

The *schedulerThread* after receiving a signal from an executing thread starts it job of picking

```
testing_scheduler()
    wait(schedulerThread. semaphore);
    while there is an enabled thread
        if i ≤ |events|
            (t_current, _, _) =event[i] ;
        else
            t_current = lowest indexed thread in the set of enabled threads;
        signal(t_current . semaphore); // release control to the thread t_current
        wait(schedulerThread. semaphore); // wait for the thread t_current to give back control
    // end of the while loop
    if there is an active thread
        print ''Error:  found deadlock'';
    compute_next_input_and_schedule() ;

access_event(m, label, access_type) // access_type can be r, w, l, u
    signal(schedulerThread. semaphore); // release control to the testing scheduler
    wait(thisThread. semaphore); // wait for the testing scheduler to give back control
    event[i] = (thisThread, label, access_type);
    enabled[i] = set of enabled threads;
    i = i + 1 ;

fork_event()
    wait(thisThread. semaphore); // wait for the testing scheduler to give back control

end_event()
    signal(schedulerThread. semaphore); // release control to the testing scheduler
```

Figure 5.3: Definition of Various Thread Execution Controlling Procedures for the Simple Testing Algorithm

the next thread to be scheduled for execution. Whenever *schedulerThread* receives a signal from a thread, it knows that all the active threads in the execution are waiting to access a shared memory location. Then it determines if there is at least one thread that is enabled among the waiting threads, that is, if there is a thread that is not waiting to acquire a lock that is already acquired by some other thread. If there is at least one enabled thread, then it picks the same thread as the previous execution while i is less than the number of elements of *event*. This ensures that the current execution follows the schedule computed in the previous execution while i is less than or equal to the length of *event*. At the end of the previous execution, the sequence *event*

```
compute_next_input_and_schedule()
    for (j = i − 1 ; j ≥ 0 ; j = j − 1)
        if event[j] is defined
            // compute a new schedule
            if |enabled[j]| > |postponed[j]| + 1
                (t, _, _) = event[j];
                postponed[j] = postponed[j] ∪ {t};
                t = smallest indexed thread in enabled[j] \ postponed[j];
                event[j] = (t, _, _) ;
                branch_hist = branch_hist[0 . . . j];
                event = event[0 . . . j];
                postponed = postponed[0 . . . j];
                return;
        else
            // compute a new input
            if (branch_hist[j].done == false)
                branch_hist[j].branch=¬branch_hist[j].branch;
                if (∃I′ that satisfies neg_last(path_c[0 . . . j]))
                    branch_hist = branch_hist[0 . . . j];
                    event = event[0 . . . j];
                    postponed = postponed[0 . . . j];
                    return;
    // end of the for loop
    if (j < 0) completed = true;
```

Figure 5.4: Compute Next Schedule or Input for the Simple Testing Algorithm

is truncated appropriately and concatenated with an event to perform a depth-first search of the feasible execution paths of P. Otherwise, if i is greater than the number of elements in *event*, *schedulerThread* selects the smallest indexed thread that it is enabled. After selecting a thread, *schedulerThread* signals the selected thread, and itself starts waiting again for a signal. If after getting a signal *schedulerThread* determines that none of the threads are enabled and there is at least one active thread in the execution, then it flags that there is a deadlock situation. Otherwise, if there is no enabled or active thread in the execution, then the program execution terminates and *schedulerThread* computes a schedule or an input for the next execution using the procedure *compute_next_input_and_schedule* described next.

5.2.3 Computing a Schedule and an Input

The procedure *compute_next_input_and_schedule* (see Figure 5.4) computes the schedule and the input that will direct the next program execution along an alternative execution path. It loops over the choice points in the current execution from the end. If the selected choice point j inside the loop contains a scheduler choice and if not all scheduler choices at the choice point have been exercised, then a new schedule is generated. Specifically, if the thread t executed at the execution point denoted by the element $event[j]$ and if t can be added to $postponed[j]$ without making $postponed[j]$ equal to the set of enabled threads at that choice point, then t is added to the set $postponed[j]$. Moreover, the smallest indexed thread, which is in the set of enabled threads at the choice point and which is not in the set $postponed[j]$, is chosen and assigned to $event[j]$. This ensures that in the next execution at the same choice point, the scheduler will pick a thread that is enabled and that is not in $postponed[j]$. Thus in subsequent executions all the threads that are enabled at the choice point will get scheduled one by one. Otherwise, if at the selected choice point $path_c[j]$ is defined and if the constraint $path_c[j]$ has not been negated previously, then constraint solving is invoked to generate a new input (see 4.1.5.)

5.3 Extending Concolic Testing with the Race-detection and Flipping Algorithm

We next show how to extend concolic testing with the race-detection and flipping algorithm. We call this combined algorithm the *efficient algorithm*. The efficient algorithm explores a much smaller superset of the execution paths in $\texttt{REx}(P)$. The algorithm accomplishes this by computing race conditions between different events in an execution. Based on these race conditions, the algorithm generates other schedules that *flip* the race conditions, to provide a depth-first search of all permutations of the race conditions in the execution path. More specifically, let $e_0 e_1 e_2 \ldots e_n$ be an execution path of a program and let e_i and e_j, where $i < j$, are related by the immediate race relation (i.e. $e_i \lessdot e_j$). In our efficient testing algorithm, we mark the event e_i (by setting $race[i]$ to true) to indicate that it has race with some future event and the thread of e_i must be postponed at that execution point in some future execution so that the race relation between e_i

and e_j gets flipped. While computing the next input and schedule at the end of the execution, if we choose to backtrack at the event e_i, then we generate a schedule for the next execution where we continue the execution up to the prefix $e_0 \dots e_{i-1}$; however, after that we postpone the execution of the thread of e_i as much as possible. This ensures that the race between e_i and e_j gets flipped or permuted (i.e. $e_j \lessdot e_i$) in the next execution and we get an execution path of the form $e_0 \dots e_{i-1} e_{i+1} \dots e_j e'_{j+1} \dots e_i \dots e'_{n'}$. For example, if $t_1 \colon x = 1, t_2 \colon x = 2$ is an execution path, then there is a race condition in the accesses of the shared variable x. We generate a schedule such that the next execution is $t_2 \colon x = 2, t_1 \colon x = 1$, i.e., the accesses to x are permuted.

In the efficient algorithm, we modify the definition of the thread controlling procedures described in Figure 5.3 by the one in Figure 5.5. (We label a statement with M: if the statement is modified or added to the pseudo-codes given in Figure 5.3.) We assume that the scheduler maintains a dynamic vector clock and a sequential vector clock with each thread and two dynamic vector clocks with each shared memory location. The dynamic vector clocks and the sequential vector clocks are updated using the procedure described in Section 3.2. These vector clocks are used to compute the $\lessdot$ relation in the procedure *check_and_set_race*. We omit the vector clock update procedures in the pseudo-code of the efficient algorithm to keep the description simple. The *access_event* procedure calls the procedure *check_and_set_race*. The procedure *check_and_set_race* determines if the current event has a race with any past event, say $event[j]$. If such a race exists, then the $race[j]$ is set to **true**—assuming that the race condition was not already flipped in a previous execution. The algorithm for selecting the next thread by the procedure *testing_scheduler* is modified so that a postponed thread's execution gets delayed as much as possible. Note that if we postpone the execution of thread as much as possible in the default schedule, then the proof of the Theorem 9 goes through, because the proof is independent of the default scheduling policy. The computation of the next input and schedule is done using the modified procedure *compute_next_input_and_schedule* (see Figure 5.6). In this procedure, a new schedule, which postpones the thread associated with an event, is generated if the event has a race with a future event. Note that in the simple algorithm (see Section 5.2), a thread is postponed at an execution point even if the corresponding event has no race with any future event.

Soundness of our algorithm is trivial; a bug reported by our algorithm is an actual bug because

```
testing_scheduler()
    wait(schedulerThread. semaphore);
    t_current = NULL;
    while there is an enabled thread
        if i ≤ |event|
            (t_current, _, _) =event[i] ;
        else
M:          if t_current is not enabled
                t_current = lowest indexed thread in the set of enabled threads;
        // otherwise schedule the thread that was scheduled in the last iteration
        signal(t_current . semaphore); // release control to the thread t_current
        wait(schedulerThread. semaphore); // wait for the thread t_current to give back control
    // end of the while loop
    if there is an active thread
        print ``Error:  found deadlock'' ;
    compute_next_input_and_schedule() ;

access_event(m, label, access_type) // access_type can be r, w, l, u
    signal(schedulerThread. semaphore); // release control to the testing scheduler
    wait(thisThread. semaphore); // wait for the testing scheduler to give back control
    event[i] = (thisThread, label, access_type);
    enabled[i] = set of enabled threads;
M:  check_and_set_race(m);
    i = i + 1 ;

fork_event()
    wait(thisThread. semaphore); // wait for the testing scheduler to give back control

end_event()
    signal(schedulerThread. semaphore); // release control to the testing scheduler

check_and_set_race(m)
    ∀j ∈ [0, i) such that event[j] ⋖ event[i]
        if t not in postponed[j]
            if e is a read or write event
                print ``Warning:  data race found'';
            race[j] = true;
```

Figure 5.5: Definition of Various Thread Execution Controlling Procedures for the Efficient Testing Algorithm

```
compute_next_input_and_schedule()
    for (j = i − 1 ; j ≥ 0 ; j = j − 1)
        if event[j] is defined
            // compute a new schedule
            if |enabled[j]| > |postponed[j]| + 1
M:              if race[j] == true
M:                  race[j] = false;
                    (t, , _) = event[j];
                postponed[j] = postponed[j] ∪ {t};
                t = smallest indexed thread in enabled[j]\ postponed[j];
                event[j] = (t, _, _) ;
                branch_hist = branch_hist[0 . . . j];
                event = event[0 . . . j];
                postponed = postponed[0 . . . j];
                return;
    else
        // compute a new input
        if (branch_hist[j].done == false)
            branch_hist[j].branch=¬branch_hist[j].branch;
            if (∃I′ that satisfies neg_last(path_c[0 . . . j]))
                branch_hist = branch_hist[0 . . . j];
                event = event[0 . . . j];
                postponed = postponed[0 . . . j];
                return;
// end of the for loop
if (j < 0) completed = true;
```

Figure 5.6: Compute Next Schedule or Input for the Efficient Testing Algorithm

our algorithm provides a concrete input and schedule on which the program exhibits the bug. Moreover, our algorithm can be complete in some cases.

Proposition 14. ***(Completeness)*** *During testing a program with our efficient algorithm, if the following conditions hold:*

- *The algorithm terminates.*
- *The algorithm makes no approximation during concolic execution and the algorithm is able to solve any constraint which is satisfiable.*

then our algorithm has executed all executions in `REx` *and we have hit all reachable statements of the program.*

```
testing_scheduler()
    wait(schedulerThread. semaphore);
    while there is an enabled thread
M:      if postponed[i] is defined
M:          delayed = delayed ∪ postponed[i] ;
M:      sleep = { nextEvent(t) | t ∈ delayed } ;
M:      t_current = smallest indexed thread from set of enabled threads \ delayed;
        signal(t_current . semaphore); // release control to the thread t_current
        wait(schedulerThread. semaphore); // wait for the thread t_current to give back control
M:      ∀e ∈ sleep if e≪ event[i − 1];
M:          let (t, _, _) = e in delayed = delayed \t ;
    // end of the while loop
    if there is an active thread
        print ``Error:  found deadlock'' ;
    compute_next_input_and_schedule() ;
```

Figure 5.7: Definition of *testing_scheduler* for the Further Optimized Algorithm

5.4 A Further Optimization

The efficient algorithm improves the simple algorithm by providing a systematic way to flip race relations between various pairs of events. However, this may result in repeated flipping of race relations between the same pair of events, if the pair of events are not next to each other. As an instance, for the example in the Section 5.3, if the next execution path is $e_0 \dots e_{i-1} e_{i+1} \dots e_j e'_{j+1} \dots e_i \dots e'_{n'}$, then our efficient algorithm may detect that there is a race between e_j and e_i. As a result our algorithm would try to flip this race once again. To avoid this, we use a technique similar to sleep sets [38]. Specifically, in the execution path $e_0 \dots e_{i-1} e_{i+1} \dots e_j e'_{j+1} \dots e_i \dots e'_{n'}$, we add the thread t, where t is the thread of the event e_i, to the set *delayed* of every event $e_{i+1}, \dots, e_j$. As a result, even if we have detected that there is a race between e_j and e_i, we would not set to true the element of *race* corresponding to the event e_j (see the 2nd line of the procedure *check_and_set_race*). This ensures that we do not repeatedly flip race relation between the same pair of events. The pseudo-code of the modified procedure *testing_scheduler* is given in Figure 5.7. The procedure *nextEvent* takes a thread t as an argument and returns the event that will happen if the thread t executes next.

5.5 Discussion

We presented an efficient algorithm for testing multi-threaded programs. An important aspect of our algorithm is that we treat symbolic constraint solving and race-flipping uniformly in our algorithm. In a given execution, we either do constraint solving or race-flipping. This helps us to test concurrent programs in a single go. A pure symbolic execution based testing algorithm for concurrent programs may end up exploring redundant execution paths having the same partial order. This is because optimal partial order reduction requires accurate knowledge of dependency relation; such knowledge may not be computable due to inaccuracies of alias analysis during symbolic execution. On the other hand, a pure concrete execution based testing algorithm for concurrent programs requires the exploration of all partial orders for all possible inputs. This may not scale up if the domain of inputs is large. Our algorithm addresses the limitations of both these approaches by combining symbolic and concrete execution. The concrete execution helps to resolve aliases exactly at runtime. As a result we get the exact dependency or the causal relation between the events. The symbolic execution helps to generate a small set of inputs from a large domain of inputs through constraint solving. Therefore, we believe that concolic testing extended with the race-detection and flipping algorithm is an attractive technique to test concurrent programs.

Chapter 6

Predictive Monitoring of Concurrent Programs

So far we described a new method of testing multi-threaded shared-memory programs. The key goal of the method was to generate test inputs and schedules so that all the reachable statements of the program are executed when the program is run on the generated inputs and schedules. As such, the proposed method can find generic bugs that are based on statement reachability. Such bugs include assertion violations, segmentation faults, uncaught exceptions, and so on. In addition, the proposed method can discover data races and deadlocks in multi-threaded shared-memory programs.

Although statement reachability-based bugs are an important class of bugs in programs, there may be bugs in a program because the program does not meet its specification. In particular, in many instances, a program may be required to satisfy a formal specification given as a formula in a suitable logic. For example, an operating system must satisfy the requirement that if there is logout by user X, then in the past the user X must have logged in. In this dissertation, we will not focus on how to determine these requirements. We will assume that in many situations a formal specification may be available along with a program. This is, in particular, often true for programs written for safety critical systems.

Given a program and its formal specification, a key research challenge is to develop scalable and automated methods either to prove that the program meets its specification or that the program has a bug with respect to the specification. A popular approach to address this problem is *model checking*. In model checking the whole state space of a program is automatically explored and checked against the formal specification. Although model checking can prove a program correct, it does not scale for large programs because the state space of practical programs is often too large to be handled by a model checker. This is called the *state explosion problem*.

Runtime verification, also called *runtime monitoring*, is an emerging approach which tries to address the state explosion problem by checking at runtime the execution of a program against

its formal specification. In particular, this approach combines testing and formal specification as follows. Given a specification, a code fragment called a *runtime monitor* is generated from the formal specification. The monitor is then weaved into the program through instrumentation so that whenever the instrumented program is executed, the monitor can check at runtime whether the specification is violated. The instrumented program is then executed on various test inputs to check if the program meets its specification on those inputs. Since testing is not rigorous, runtime monitoring can find violations of a specification a program, but it cannot prove that a program meets its specification.

We can combine runtime monitoring with concolic testing to test a sequential program against its formal specification. In the ideal case, if concolic testing manages to explore all feasible paths of a program and runtime monitoring does not detect any violation of the specification in the explored paths, then we can prove that the program meets its specification.

However, if our program is a shared-memory multi-threaded program, then a combination of runtime monitoring and concolic testing extended with the race-detection and flipping algorithm is not sufficient, that is, even if we find no violation of the specification by exploring one execution path from each feasible partial order of the program, there may exist unexplored feasible execution paths that may violate the property. This is because exploring only non-equivalent execution paths is not sufficient for catching violations of temporal properties—a temporal property may be simultaneously satisfied and violated by two different equivalent execution paths. This was illustrated by an example in Section 2.3.

Next we present a technique to address the above problem. The technique is called *predictive monitoring*.[1] In predictive monitoring, from an observed execution path, we generate all the equivalent execution paths and represent them compactly in an abstract model called *computation lattice*. We show that runtime monitoring on this model can be done efficiently. Since this technique enables us to predict violations of properties in non-observed execution paths without re-executing the program, we call the technique *predictive*. Observe that predictive monitoring can predict and monitor all execution paths equivalent to a given execution path; therefore, predictive monitoring is not comprehensive like model checking. However, when combined with concolic testing, predictive

[1]This work is done partly in collaboration with Gul Agha and Grigore Roşu. Part of this work appeared in [90, 92, 80, 81, 94].

monitoring makes the former more efficient because it does not re-execute the program along equivalent paths, but relies only on the information that is already available from an execution.

The rest of the chapter is organized as follows. In Section 6.1, we describe a simple form of monitors for safety properties. In Section 6.2, we define relevant causality which is a refinement of the notion of causality relation described in Section 3.2. We describe the predictive monitoring algorithm in Section 6.3.

6.1 Monitors for Safety Properties

Safety properties are a very important, if not the most important, class of properties that one should consider in monitoring. This is because once a system violates a safety property, there is no way to continue its execution to satisfy the safety property later. Therefore, a monitor for a safety property can precisely say at runtime when the property has been violated, so that an external recovery action can be taken. From a monitoring perspective, what is needed from a safety formula is a succinct representation of its *bad prefixes* which are finite sequences of states leading to a violation of the property. Therefore, one can abstract away safety properties by languages over finite words.

Automata are a standard means to succinctly represent languages over finite words. In what follows we define a suitable version of automata, called *monitor*, with the property that it has a "bad" state from which it never gets out:

Definition: Let $\mathcal{S}$ be a finite or infinite set, that can be thought of as the set of states of the program to be monitored. Then an *$\mathcal{S}$-monitor* or simply a *monitor*, is a tuple $\mathcal{M}on = \langle \mathcal{M}, m_0, b, \rho \rangle$, where

- $\mathcal{M}$ is the set of states of the monitor;
- $m_0 \in \mathcal{M}$ is the initial state of the monitor;
- $b \in \mathcal{M}$ is the *final state* of the monitor, also called *bad state*; and
- ρ: $\mathcal{M} \times \mathcal{S} \to 2^{\mathcal{M}}$ is a transition function with the property that $\rho(b, \Sigma) = \{b\}$ for any $\Sigma \in \mathcal{S}$.

Sequences in $\mathcal{S}^\star$, where ϵ is the empty one, are called *(execution) traces*. A trace π is said to be a *bad prefix in* $\mathcal{M}$ *on* iff $b \in \rho(\{m_0\}, \pi)$, where $\rho\colon 2^{\mathcal{M}} \times \mathcal{S}^\star \to 2^{\mathcal{M}}$ is recursively defined as $\rho(M, \epsilon) = M$ and $\rho(M, \pi\Sigma) = \rho(\rho(M, \pi), \Sigma)$, where $\rho\colon 2^{\mathcal{M}} \times \mathcal{S} \to 2^{\mathcal{M}}$ is defined as $\rho(\{m\} \cup M, \Sigma) = \rho(m, \Sigma) \cup \rho(M, \Sigma)$ and $\rho(\emptyset, \Sigma) = \emptyset$, for all finite $M \subseteq \mathcal{M}$ and $\Sigma \in \mathcal{S}$.

$\mathcal{M}$ is not required to be finite in the above definition, but $2^{\mathcal{M}}$ represents the set of *finite* subsets of $\mathcal{M}$. In practical situations it is often the case that the monitor is *not* explicitly provided in a mathematical form as above. For example, a monitor can be a specific type of program whose execution is triggered by receiving events from the monitored program; its state can be given by the values of its local variables, and the bad state is a fixed unique state which once reached cannot be changed by any further events.

There are fortunate situations in which monitors can be *automatically generated* from formal specifications, thus requiring the user to focus on system's formal safety requirements rather than on low level implementation details. In fact, this was the case in all the experiments that we have performed so far. We have so far experimented with requirements expressed either in extended regular expressions (ERE) or various variants of temporal logics, with both future and past time operators. For example, [88, 89] show coinductive techniques to generate minimal static monitors from EREs and from future time linear temporal logics, respectively, and [50, 13] show how to generate dynamic monitors, i.e., monitors that generate their states on-the-fly, as they receive the events, for the safety segment of temporal logic. Note, however, that there may be situations in which the generation of a monitor may not be feasible, even for simple requirements languages. For example, it is well-known that the equivalent automaton of an ERE may be non-elementary larger than the ERE in the worst case [105]; therefore, there exist relatively small EREs whose monitors cannot even be stored.

Example 15. *Consider a reactive controller that maintains the water level of a reservoir within safe bounds. It consists of a water level reader and a valve controller. The water level reader reads the current level of the water, calculates the quantity of water in the reservoir and stores it in a shared variable* `w`*. The valve controller controls the opening of a valve by looking at the current quantity of water in the reservoir. A very simple and naive implementation of this system contains two threads:* `T1`*, the valve controller, and* `T2`*, the water level reader. The code snippet is given in*

```
Thread T1:

while(true) {
  if(w > 18) delta = 10;
  else delta = -10;
  for(i=0; i<2; i++) {
    v = v + delta;
    setValveOpening(v);
    sleep(100);
  }
}
```

```
Thread T2:

while(true) {
  l = readLevel();
  w = calcVolume(l);
  sleep(100);
}
```

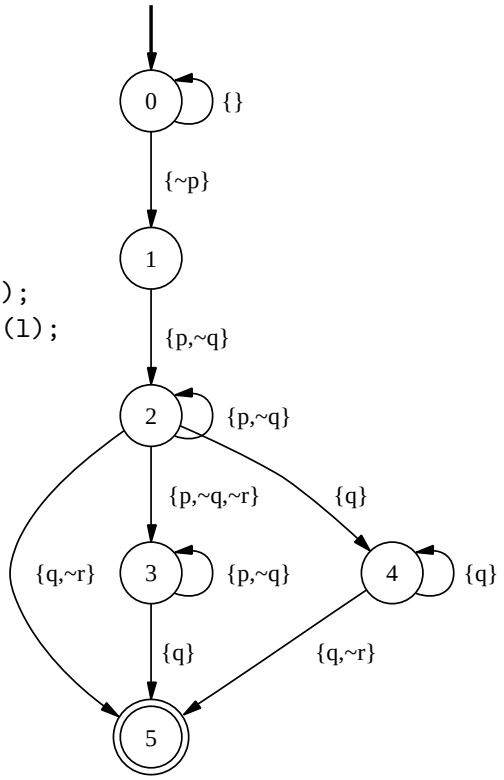

Figure 6.1: Two Threads (`T1` controls the valve and `T2` reads the water level) and a Monitor.

Figure 6.1.

Here `w` *is in some proper units such as mega gallons and* `v` *is in percentage. The implementation is poorly synchronized and it relies on ideal thread scheduling.*

A sample run of the system can be $\{w = 20, v = 40\}, \{w = 24\}, \{v = 50\}, \{w = 27\}, \{v = 60\}, \{w = 31\}, \{v = 70\}$*. As we will see later in the paper, by a run we here mean a sequence of relevant variable writes. Suppose we are interested in a safety property that says "If the water quantity is more than 30 mega gallons, then it is the case that sometime in the past water quantity exceeded 26 mega gallons and since then the valve is open by more than 55% and the water quantity never went down below 26 mega gallon". We can express this safety property in two different formalisms: linear temporal logic (LTL) with both past-time and future-time operators, or extended regular expressions (EREs) for bad prefixes. The atomic propositions that we will consider are* $p : (w > 26), q : (w > 30), r : (v > 55)$*. The properties can be written as follows:*

$$
\begin{aligned}
F_1 &= \Box(q \rightarrow ((r \wedge p)\mathcal{S} \uparrow p)) \\
F_2 &= \{\}^*\{\neg p\}\{p, \neg q\}^+ \\
&\quad (\{p, \neg q, \neg r\}\{p, \neg q\}^*\{q\} + \{q\}^*\{q, \neg r\})\{\}^*
\end{aligned}
$$

The formula F_1 in LTL ($\uparrow p$ is a shorthand for "p and previously not p") states that "It is always the case that if $(w > 30)$, then at some time in the past $(w > 26)$ started to be true and since then $(r > 55)$ and $(w > 26)$." The formula F_2 characterizes the prefixes that make F_1 false. In F_2 we use $\{p, \neg q\}$ to denote a state where p and $\neg q$ holds and r may or may not hold. Similarly, $\{\}$ represents any state of the system. The monitor automaton for F_2 is given also in Figure 6.1.

6.2 Relevant Causality

Monitors for a multi-threaded program may refer to a small subset of the set of the variables of the program. In our technique, we restrict such variables to a set called *path-robust* variables. A variable is said to be *path-robust* if and only if its value remains the same along an execution path, that is, its value is independent of the input along any execution path. However, the value of such a variable may be different for different executions paths of the program. A path-robust variable is said to be a *relevant variable*, if it is referred in a monitor of the program.

For example in the multi-threaded program in Figure 2.3, the set of path-robust variables is {`x`, `y` }. Since the value of `z` may vary along a particular execution path depending on the input, `z` is not a path-robust variable. If we consider a monitor for the property ``` ``Always x greater than 0,'' ``` then the monitor only refers to the path-robust variable `x`. All the other variables in the program except x are essentially irrelevant for the monitor. Therefore, the set of relevant variables is { `x` }.

To minimize the number of messages, like in [65] which suggests a similar technique but for distributed systems in which reads and writes are not distinguished, we consider a subset $\mathcal{R} \subseteq \mathcal{E}$ of *relevant events*. We say that $e \in E$ is a relevant event if and only if e writes a memory location m, x is a relevant variable, and m is the address of x (i.e. $m = \&x$). We define the *$\mathcal{R}$-relevant causality* on $\mathcal{E}$ as the relation $\preccurlyeq_{\mathcal{R}} := \preccurlyeq \cap (\mathcal{R} \times \mathcal{R})$, so that $e \preccurlyeq_{\mathcal{R}} e'$ iff $e, e' \in \mathcal{R}$ and $e \preccurlyeq e'$. It is important to notice though that the other variables can also indirectly influence the relation $\preccurlyeq_{\mathcal{R}}$,

because they can influence the relation $\preccurlyeq$. We next provide a technique based on *dynamic vector clocks* that correctly implements the relevant causality relation.

6.2.1 Vector Clock Algorithm for Relevant Causality

We provide a variant of the dynamic vector clock algorithm (see Section 3.2), called the *relevant vector clock algorithm*, that correctly and efficiently implements the relevant causality relation.

The relevant causality relation between the events in an execution can be tracked efficiently at runtime using *relevant vector clocks* (RVC). A relevant vector clock $VR\colon T \to \mathbb{N}$, where T is the set of threads that are present in the execution. We call such a map a *relevant vector clock (RVC)*. Such a map can be partial because threads are created dynamically at runtime. To simplify the exposition and the implementation, we assume that each RVC VR is a total map, where $VR(t) = 0$ whenever VR is not defined on thread t.

We associate a RVC with every thread t and denote it by VR_t. Moreover, we associate two RVCs VR^a_m and VR^w_m with every shared memory m; we call the former *access RVC* and the latter *update RVC*. Whenever a thread t with current RVC VR_t generates an event e, the following algorithm, called the *relevant vector clock algorithm*, is executed:

1. If e is a relevant event, i.e., if $e \in \mathcal{R}$, then

   $VR_t(t) \leftarrow VR_t(t) + 1$.

2. If e is a read of a shared memory location m, then

   $VR_t \leftarrow \max\{\, VR_t,\, VR^w_m \}$

   $VR^a_m \leftarrow \max\{\, VR^a_m,\, VR_t \}$

3. If e is a write, lock, or unlock of a shared memory location m, then

   $VR^w_m \leftarrow VR^a_m \leftarrow VR_t \leftarrow \max\{\, VR^a_m,\, VR_t \}$

4. If e is a fork event and if t' is the newly created thread, then

   $VR_{t'} \leftarrow VR_t$

5. If e is a relevant event and if e writes the value v to the variable x, then

   send message $\langle t,\, VR_t, x, v \rangle$.

We can associate a RVC with every event e, denoted by $VR\{e\}$, as follows. If e is executed by t and if VR_t is the vector clock of t just after the event e, then $VR\{e\} = VR_t$. Given a multi-threaded program, we instrument the program so that it runs the relevant vector clock algorithm for every event and sends the messages to an observer which performs predictive monitoring.

Lemma 16. *After the processing of the event e_t^k by thread t*

(a) $VR_t(t')$ equals the number of relevant events of t that causally precede e_t^k.

(b) $VR_m^a(t')$ equals the number of relevant events of t that causally precede any event in E that appears before or equals to e_t^k and accessed m.

(c) $VR_m^w(t')$ equals the number of relevant events of t' that causally precede the most recent write event of m.

Theorem 17. *For any two events e and e', $e \preccurlyeq_{\mathcal{R}} e'$ iff $VR\{e\} \leq VR\{e'\}$.*

The proof of the above two results can be done in a way similar to that of the Theorem 6.

6.3 Runtime Model Generation and Predictive Monitoring

In an execution of a multi-threaded program, the messages sent by the relevant vector clock algorithm are received by an observer which performs the predictive monitoring. The observer receives messages of the form $\langle t, VR_t, x, v\rangle$. Each such message represents an event on thread t whose RVC is VR_t, and in that event the value v has been assigned to the relevant variable x. Because of Theorem 17, the observer can infer the causal dependency between the relevant events emitted by the execution of the multi-threaded program. We show how the observer can monitor all possible interleavings of events that do not violate the observed causal dependency. Only one of these interleavings corresponds to the real execution, the others being all potential executions. Hence, the presented technique can *predict* safety violations from successful executions.

6.3.1 Multi-Threaded Computation Lattice

Inspired by related definitions in [9], we define the important notions of relevant multi-threaded computation and run as follows. A *relevant multi-threaded computation*, simply called *multi-*

threaded computation from now on, is the partial order on the relevant events that the observer can infer, which is nothing but the relation $\preccurlyeq_{\mathcal{R}}$. A *relevant multi-threaded run*, also simply called *multi-threaded run* from now on, is any permutation of the relevant events, which *does not violate* the multi-threaded computation. Our goal is to check safety requirements against *all* (relevant) multi-threaded runs of a multi-threaded computation.

A relevant event can change the state of the multi-threaded program as seen by the observer; this is formalized next. A *relevant program state*, or simply a *program state*, is a map from relevant variables to concrete values. Any permutation of relevant events generates a sequence of program states in the obvious way, however, not all permutations of relevant events are valid multi-threaded runs. A program state is called *consistent* if and only if there is a multi-threaded run containing that state in its sequence of generated program states. We next formalize these concepts.

Definition: [Consistent Run] For a given permutation of events in $\mathcal{R}$, say $R = e_1 e_2 \ldots e_{|\mathcal{R}|}$, we say that R is a *consistent run* if for all pairs of events e and e', $e \preccurlyeq_{\mathcal{R}} e'$ implies that e appears before e' in R.

Let e_t^k be the k^{th} relevant event generated by the thread t since the start of its execution. A *cut* C is a subset of $\mathcal{R}$ such that for all $t \in T$ if e_t^k is present in C, then for all $l < k$, e_t^l is also present in C. In what follows, we let $T = \{t_1, \ldots, t_n\}$ to be the set of all threads created during the execution. A cut is denoted by a tuple $(e_{t_1}^{k_1}, e_{t_2}^{k_2}, ..., e_{t_n}^{k_n})$ where each entry in the tuple corresponds to the last relevant event for each thread included in C. If a thread t has not seen a relevant event, then the corresponding entry is denoted by e_t^0. A cut C corresponds to a relevant program state that has been reached after all the events in C have been executed. Such a relevant program state is called a *relevant global multi-threaded state*, or simply a *relevant global state* or even just *state*, and is denoted by $\Sigma^{k_1 k_2 \ldots k_n}$.

Definition: [Consistent Cut] A cut is said to be consistent if for all events e and e'

$$(e \in C) \wedge (e' \preccurlyeq_{\mathcal{R}} e) \rightarrow (e' \in C)$$

A consistent global state is the one that corresponds to a consistent cut. A relevant event e_t^l is said to be enabled in a consistent global state $\Sigma^{k_1 k_2 \ldots k_n}$ if and only if $C \cup \{e_t^l\}$ is a consistent cut,

where C is the consistent cut corresponding to the state $\Sigma^{k_1k_2\ldots k_n}$. The following proposition holds for an enabled event:

Proposition 18. *A relevant event $e^l_{t_i}$ is enabled in a consistent global state $\Sigma^{k_1k_2\ldots k_n}$ if and only if $l = k_i + 1$. Moreover, for all relevant events e, if $e \neq e^l_{t_i}$ and $e \preccurlyeq_{\mathcal{R}} e^l_{t_i}$, then $e \in C$, where C is the consistent cut corresponding to the state $\Sigma^{k_1k_2\ldots k_n}$.*

Proof. Since $e^l_{t_i}$ is enabled in the state $\Sigma^{k_1k_2\ldots k_n}$, $C \cup \{e^l_{t_i}\}$ is a cut. This implies that for all events $e^k_{t_i}$, if $k < l$, then $e^k_{t_i} \in C \cup \{e^l_{t_i}\}$ and hence $e^k_{t_i} \in C$. In particular, all the events $e^1_{t_i}, e^2_{t_i}, \ldots, e^{l-1}_{t_i}$ are in C. However, $e^{l-1}_{t_i}$ is the last relevant event from thread t_i, which is included in C. Therefore, $k_i = l - 1$. The other way follows trivially because $e^{k_i}_{t_i} \preccurlyeq_{\mathcal{R}} e^l_{t_i}$ and $e \preccurlyeq_{\mathcal{R}} e^{k_i}_{t_i}$ for all $e \in C$.

Since $e^l_{t_i} \in C \cup \{e^l_{t_i}\}$, $e \preccurlyeq_{\mathcal{R}} e^l_{t_i}$, and $C \cup \{e^l_{t_i}\}$ is a consistent cut, $e \in C \cup \{e^l_i\}$ (by the definition of consistent cut). Since by assumption $e \neq e^l_{t_i}$, we have $e \in C$. □

An immediate consequence of the above proposition is the following corollary:

Corollary 19. *If C is the consistent cut corresponding to the state $\Sigma^{k_1k_2\ldots k_n}$ and if $e^l_{t_i}$ is enabled in $\Sigma^{k_1k_2\ldots k_n}$, then the state corresponding to the consistent cut $C \cup \{e^l_{t_i}\}$ is $\Sigma^{k_1k_2\ldots k_{i-1}lk_{i+1}\ldots k_n}$ or $\Sigma^{k_1k_2\ldots k_{i-1}(k_i+1)k_{i+1}\ldots k_n}$ and we denote it by $\delta(\Sigma^{k_1k_2\ldots k_n}, e^l_{t_i})$.*

Here the partial function δ maps a consistent state Σ and a relevant event e enabled in that state to a consistent state $\delta(\Sigma, e)$ which is the result of executing e in Σ. Let Σ^{K_0} be the *initial* global state, $\Sigma^{00\ldots0}$, which is always consistent. The following result holds:

Lemma 20. *If $R = e_1e_2\ldots e_{|\mathcal{R}|}$ is a consistent multi-threaded run, then it generates a sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|\mathcal{R}|}}$ such that for all $r \in [1, |\mathcal{R}|]$, $\Sigma^{K_{r-1}}$ is consistent, e_r is enabled in $\Sigma^{K_{r-1}}$, and $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$.*

Proof. The proof is by induction on r. By definition the initial state Σ^{K_0} is consistent. Moreover, e_1 is enabled in Σ^{K_0} because the cut C corresponding to the state Σ^{K_0} is empty and hence the cut $C \cup \{e_1\} = \{e_1\}$ is consistent. Since Σ^{K_0} is consistent and e_1 is enabled in Σ^{K_0}, $\delta(\Sigma^{K_0}, e_1)$ is defined. Let $\Sigma^{K_1} = \delta(\Sigma^{K_0}, e_1)$.

Let us assume that $\Sigma^{K_{r-1}}$ is consistent, e_r is enabled in $\Sigma^{K_{r-1}}$, and $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$. Therefore, $\delta(\Sigma^{K_{r-1}}, e_r) = \Sigma^{K_r}$ is also consistent. Let C be the cut corresponding to Σ^{K_r}. To prove

that e_{r+1} is enabled in Σ^{K_r} we have to prove that $C \cup \{e_{r+1}\}$ is a cut and it is consistent. Let $e_{r+1} = e_t^l$ for some t and l i.e. e_{r+1} is the l^{th} relevant event of thread t. For every event e_t^k such that $k < l$, $e_t^k \preccurlyeq_{\mathcal{R}} e_t^l$. Therefore, by the definition of consistent run, e_t^k appears before e_t^l in R for all $0 < k < l$. This implies that all e_t^k for $0 < k < l$ are included in C. This proves that $C \cup e_t^l$ is a cut. Since C is a cut, for all events e and e' if $e \neq e_t^l$, then $(e \in C \cup \{e_t^l\}) \wedge (e' \preccurlyeq_{\mathcal{R}} e) \rightarrow e' \in C \cup \{e_t^l\}$. Otherwise, if $e = e_t^l$, then by the definition of consistent run, if $e' \preccurlyeq_{\mathcal{R}} e_t^l$, then e' appears before e_t^l in R. This implies that e' is included in $C \cup \{e_t^l\}$. Therefore, $C \cup \{e_t^l\}$ is consistent, which proves that $e_{r+1} = e_t^l$ is enabled in the state Σ^{K_r}. Since Σ^{K_r} is consistent and e_{r+1} is enabled in Σ^{K_r}, $\delta(\Sigma^{K_r}, e_{r+1})$ is defined. We let $\delta(\Sigma^{K_r}, e_{r+1}) = \Sigma^{K_{r+1}}$. □

From now on, we identify the sequences of states $\Sigma^{K_0}\Sigma^{K_1}\ldots\Sigma^{K_{|\mathcal{R}|}}$ as above with multi-threaded runs, and simply call them *runs*. We say that Σ *leads-to* Σ', written $\Sigma \rightsquigarrow \Sigma'$, when there is some run in which Σ and Σ' are consecutive states. Let $\rightsquigarrow^*$ be the reflexive transitive closure of the relation $\rightsquigarrow$. The set of all consistent global states together with the relation $\rightsquigarrow^*$ forms a *lattice* with n mutually orthogonal axes representing each thread. For a state $\Sigma^{k_1k_2\ldots k_n}$, we call $k_1 + k_1 + \cdots k_n$ its *level*. A *path* in the lattice is a sequence of consistent global states on increasing level, where the level increases by 1 between any two consecutive states in the path. Therefore, a run is just a path starting with $\Sigma^{00\ldots0}$ and ending with $\Sigma^{r_1r_2\ldots r_n}$, where r_i is the total number of relevant events of thread t_i.

Therefore, a multi-threaded computation can be seen as a lattice. This lattice, which is called *computation lattice* and referred to as $\mathcal{L}$, should be seen as an *abstract model* of the running multi-threaded program, containing the relevant information needed in order to analyze the program. Supposing that one is able to *store* the computation lattice of a multi-threaded program, which is a non-trivial matter because it can have an exponential number of states in the length of the execution, one can mechanically model-check it against the safety property.

Given a state $\Sigma^{k_1k_2\ldots k_n}$ we can associate a RVC with the state (denoted by $VR\{\Sigma^{k_1k_2\ldots k_n}\}$) such that $VR\{\Sigma^{k_1k_2\ldots k_n}\}(t_i) = k_i$ i.e. $VR\{\Sigma^{k_1k_2\ldots k_n}\}(t_i)$ is equal to the number of relevant events of thread t_i that has causally effected the state. With this definition the following results hold:

Lemma 21. *If a relevant event e from a thread t is enabled in a state Σ and if $\delta(\Sigma, e) = \Sigma'$, then $\forall t' \neq t$: $VR\{\Sigma\}(t') = VR\{\Sigma'\}(t')$ and $VR\{\Sigma\}(t) + 1 = VR\{\Sigma'\}(t)$.*

Proof. This follows directly from the definition of RVC of a state and Corollary 19. □

Lemma 22. *If a relevant event e from thread t is enabled in a state Σ, then $\forall t' \neq t$: $VR\{\Sigma\}(t') \geq VR\{e\}(t')$ and $VR\{\Sigma\}(t) + 1 = VR\{e\}(t)$.*

Proof. $VR\{\Sigma\}(t) + 1 = VR\{e\}(t)$ follows from Lemma 21. Say $k = VR\{e\}(t')$ for some $t' \neq t$. Then by (a) of Lemma 16 we know that the k^{th} relevant event from thread t' causally precedes e i.e. $e^k_{t'} \preccurlyeq_{\mathcal{R}} e$. Then by proposition 18, $e^k_{t'} \in C$, where C is the cut corresponding to Σ. This implies that $k \leq VR\{\Sigma\}(t')$ which proves that $\forall t' \neq t$: $VR\{\Sigma\}(t') \geq VR\{e\}(t')$. □

Lemma 23. *If $R = e_1 e_2 \dots e_{|\mathcal{R}|}$ is a consistent multi-threaded run generating the sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\dots\Sigma^{K_{|\mathcal{R}|}}$, then $VR\{\Sigma^{K_i}\}$ can be recursively defined as follows:*

$$VR\{\Sigma^{K_0}\}(t) = 0 \qquad \text{for all } t \in T$$
$$VR\{\Sigma^{K_r}\}(t) = max(VR\{\Sigma^{K_{r-1}}\}(t), VR\{e_r\}(t)) \quad \text{for all } t \in T \text{ and } 0 < r \leq |\mathcal{R}|$$

Proof. $\forall t \in T$: $VR\{\Sigma^{K_0}\}(t) = 0$ holds by definition. Let e_r be from thread t'. By Lemma 20 e_r is enabled in $\Sigma^{K_{r-1}}$. Therefore, by Lemma 22, $\forall t \neq t'$: $VR\{\Sigma^{K_{r-1}}\}(t) \geq VR\{e_r\}(t)$. This implies that $\forall t \neq t'$: $VR\{\Sigma^{K_r}\}(t) = VR\{\Sigma^{K_{r-1}}\}(t) = max(VR\{\Sigma^{K_{r-1}}\}(t), VR\{e_r\}(t))$. Otherwise if $t = t'$, by Lemma 22, $VR\{\Sigma^{K_{r-1}}\}(t)+1 = VR\{e_r\}(t)$. Therefore, $VR\{\Sigma^{K_r}\}(t) = VR\{\Sigma^{K_{r-1}}\}(t)+1 = VR\{e_r\}(t) = max(VR\{\Sigma^{K_{r-1}}\}(t), VR\{e_r\}(t))$. This proves that $\forall j \in T$: $VR\{\Sigma^{K_r}\}(t) = max(VR\{\Sigma^{K_{r-1}}\}(t), VR\{e_r\}(t))$. □

Corollary 24. *If $R = e_1 e_2 \dots e_{|\mathcal{R}|}$ is a consistent multi-threaded run generating the sequence of global states $\Sigma^{K_0}\Sigma^{K_1}\dots\Sigma^{K_{|\mathcal{R}|}}$, then*

$$VR\{\Sigma^{K_r}\}(t) = max(VR\{e_1\}(t), VR\{e_2\}(t), \dots, VR\{e_r\}(t)) \quad \text{for all } t \in T \text{ and } 0 < r \leq |\mathcal{R}|$$

Example 25. *Figure 6.2 shows the causal partial order on relevant events extracted by the observer from the multi-threaded execution in Example 15 together with the generated computation lattice. The actual execution, $\Sigma^{00}\Sigma^{01}\Sigma^{11}\Sigma^{12}\Sigma^{22}\Sigma^{23}\Sigma^{33}$, is marked with solid edges in the lattice. Besides its RVC, each global state in the lattice stores its values for the relevant variables w and v. It can be readily seen from Figure 2.3 that the LTL property F_1 defined in Example 15 holds on the*

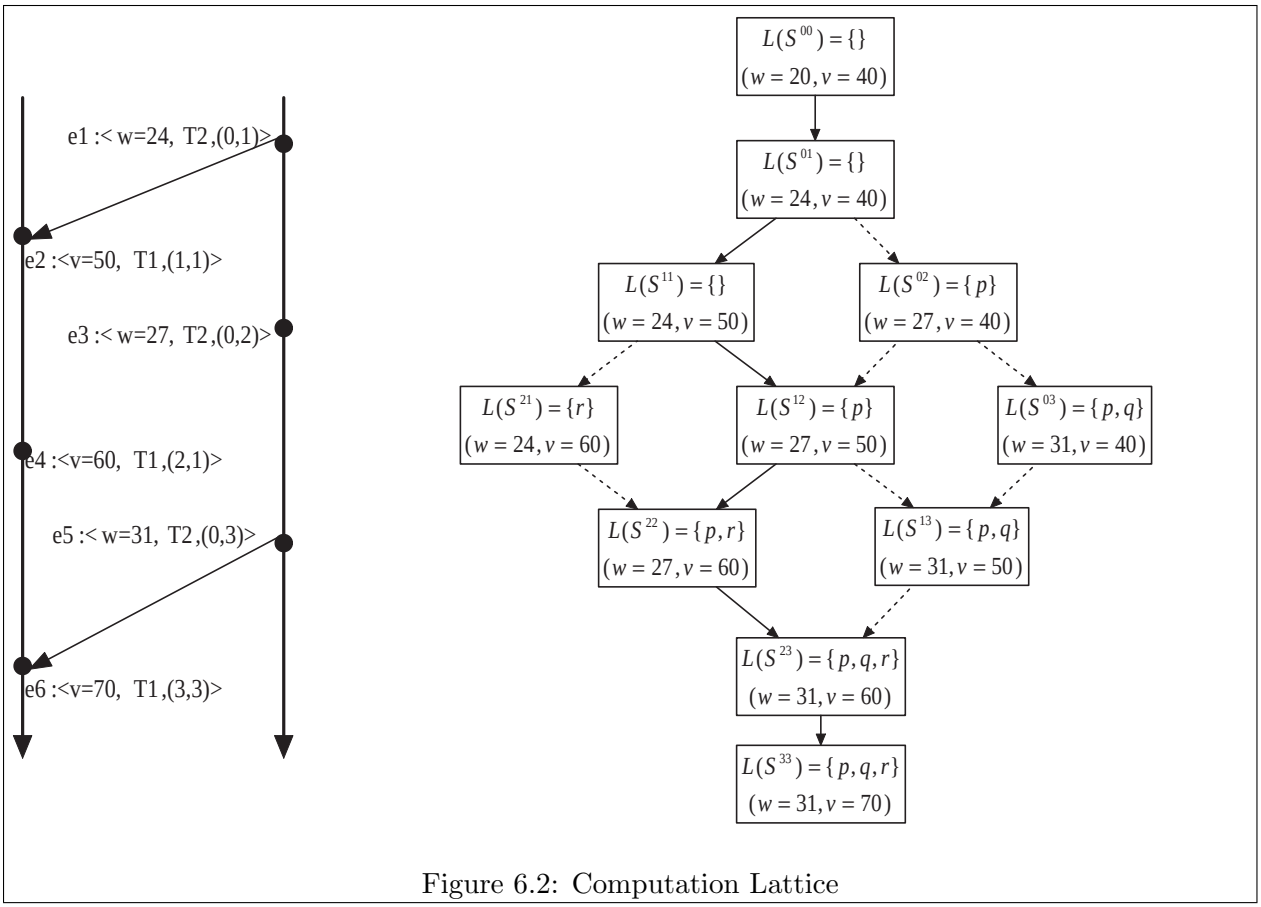

Figure 6.2: Computation Lattice

sample run of the system, and also that it is not in the language of bad prefixes, F_2*. However,* F_1 *is violated on some other consistent runs, such as* $\Sigma^{00}\Sigma^{01}\Sigma^{02}\Sigma^{12}\Sigma^{13}\Sigma^{23}\Sigma^{33}$*. On this particular run* $\uparrow p$ *holds at* Σ^{02}*; however,* r *does not hold at the next state* Σ^{12}*. This makes the formula* F_1 *false at the state* Σ^{13}*. The run can also be symbolically written as* $\{\}\{\}\{p\}\{p\}\{p,q\}\{p,q,r\}\{p,q,r\}$*. In the automaton in Figure 6.1, this corresponds to a possible sequence of states* 00123555*. Hence, this string is accepted by* F_2 *as a bad prefix.*

Therefore, by carefully analyzing the computation lattice extracted from a successful execution one can infer safety violations in other possible consistent executions. In what follows we propose effective techniques to analyze the computation lattice. A first important observation is that one can generate it *on-the-fly* and analyze it on a level-by-level basis, discarding the previous levels. However, even if one considers only one level, that can still contain an exponential number of states in the length of the current execution. A second important observation is that the states in the computation lattice are not all equiprobable in practice. By allowing an user configurable *window*

of most likely states in the lattice centered around the observed execution trace, the presented technique becomes quite scalable, requiring $O(wm)$ space and $O(twm)$ time, where w is the size of the window, m is the size of the bad prefix monitor of the safety property, and t is the size of the monitored execution trace.

6.3.2 Level by Level Analysis of the Computation Lattice

Given an execution path τ of a multi-threaded program, the relevant vector clock algorithm generates a message for each relevant event in τ. The purpose of the predictive monitoring is to check all execution paths in $[\tau]_{\equiv_{\preccurlyeq}}$ against a given monitor. The check is performed by an observer which has access to the sequence of messages generated from τ.

A naïve observer would just check the observed sequence of messages against the monitor for the safety property, say $\mathcal{M}on$ like in Definition 6.1, and would maintain at each moment a set of states, say *MonStates* in $\mathcal{M}$. Let Q be the sequence of messages received from the execution τ. For each message in the sequence, it would create the next state Σ and replace *MonStates* by $\rho(MonStates, pgmState(\Sigma))$, where $pgmState(\Sigma)$ gives the mapping of all relevant program variables to their values in the state Σ. If the bad state b will ever be in *MonStates*, then a property violation error would be reported, meaning that the current execution trace led to a bad prefix of the safety property. Here we assume that the messages are received in the order in which they are emitted, and also that the monitor works over the global states of the multi-threaded programs. This assumption is essential for the observer to deduce the actual execution path of the multi-threaded program. The knowledge of the actual execution path is used by the observer to apply the causal cone heuristics as described later. The assumption is not necessary if we do not want to use causal cone heuristics. The work in [91] describes a technique for the level by level analysis of the computation lattice without the above assumption.

The pseudo-code for the naïve observer is given in Figure 6.3. $pgmState(\Sigma)[x \mapsto v]$ is same as $pgmState(\Sigma)$, except the value of the relevant variable x being v.

A smart observer, as said before, will analyze not only the observed execution trace, but also all the other consistent runs of the multi-threaded system, thus being able to *predict* violations from successful executions. The observer receives the messages from the running multi-threaded

Let $\mathcal{M}on = \langle \mathcal{M}, m_0, b, \rho \rangle$ be the given monitor;
$pgmState(\Sigma^{K_0})$ maps each relevant variable to its initial value;
$MonStates(\Sigma^{K_0}) = \rho(\{m_0\}, pgmState(\Sigma^{K_0}))$;
global var $CurrentState = \Sigma^{K_0}$;

// inputs: Q is the sequence of messages received from the program execution
$runNaïveObserver(Q)$
    for each $msg \in Q$
        $\langle t, VR, x, v \rangle = msg$;
        $NextState = pgmState(CurrentState)[x \mapsto v]$;
        $MonStates(NextState) = \rho(MonStates(CurrentState),\ pgmState(NextState))$;
        if $b \in MonStates(NextState)$;
            print "property violated";
        $CurrentState = NextState$;

Figure 6.3: Monitoring a Linear Trace

program and appends them to the message sequence Q. At the end of the execution, it traverses the computation lattice level by level and checks whether the bad state of the monitor can be hit by any of the runs up to the current level. We next provide the algorithm that the observer uses to construct the lattice level by level from the sequence of messages Q it receives from the execution path τ of an instrumented program.

The observer maintains a list of global states (*CurrLevel*), that are present in the current level of the lattice. For each message *msg* in Q, it tries to construct a new global state from the set of states in the current level and the message *msg*. If the global state is created successfully, then it is added to the list of global states (*NextLevel*) for the next level of the lattice. The process continues until certain condition, *levelComplete?()* holds. At that time the observer says that the level is complete and starts constructing the next level by setting *CurrLevel* to *NextLevel*, *NextLevel* to empty set, and reallocating the space previously occupied by *CurrLevel*. Here the predicate *levelComplete?()* is crucial for generating only those states in the level that are most likely to occur in other executions, namely those in the *window*, or the *causality cone*, that is described in the next subsection. The *levelComplete?* predicate is also discussed and defined in the next subsection. The pseudo-code for the level-by-level monitoring of the lattice is given in Figure 6.4.

Every global state Σ contains the value of all relevant shared variables in the program, a RVC

Let $\mathcal{M}on = \langle \mathcal{M}, m_0, b, \rho \rangle$ be the given monitor;
$pgmState(\Sigma^{K_0})$ maps each relevant variable to its initial value;
$MonStates(\Sigma^{K_0}) = \rho(\{m_0\}, pgmState(\Sigma^{K_0}))$;
$\forall t \in T,\ VR\{\Sigma^{K_0}\}(t) = 0$;
global var $CurrLevel = \{\Sigma^{K_0}\}$;
global var $NextLevel = \emptyset$;

// inputs: Q is the sequence of messages received from the program execution
runObserver(Q)
    while Q is not empty
        $Q =$ *constructLevel*(Q);

constructLevel(Q)
    for each $msg \in Q$ **and** $\Sigma \in CurrLevel$
        if *nextState?*$(\Sigma,\ msg)$
            $NextLevel = NextLevel \uplus createState(\Sigma,\ msg)$;
            if *levelComplete?*$(NextLevel,\ msg, Q)$
                $Q = removeUselessMessages(CurrLevel, Q)$;
                $CurrLevel = NextLevel$;
                $NextLevel = \emptyset$;
                return Q;

nextState?$(\Sigma,\ msg\ )$
    $\langle t,\ VR, x, v \rangle = msg$;
    if $(\forall t' \neq t : VR\{\Sigma\}(t') \geq VR(t')$ **and** $VR\{\Sigma\}(t) + 1 == VR(t))$
        return true;
    return false;

createState$(\Sigma,\ msg\ )$
    $\langle t,\ VR, x, v \rangle = msg$;
    $\Sigma' =$ new copy of Σ;
    $VR\{\Sigma'\}(t) = VR\{\Sigma\}(t) + 1$;
    $pgmState(\Sigma')[x \mapsto v]$;
    $MonStates(\Sigma') = \rho(MonStates(\Sigma), pgmState(\Sigma'))$;
    if $b \in MonStates(\Sigma')$;
        print "property violated";
    return Σ';

Figure 6.4: Level-by-level Monitoring of a Computation Lattice

$VR\{\Sigma\}$ to represent the latest events from each thread that resulted in that global state. Here the predicate *nextState?*(Σ, *msg*), checks if the event corresponding to the message *msg* is enabled in the state Σ. The correctness of the predicate is given by Lemma 22. It says that event e can generate a consecutive state for a state Σ, if and only if Σ 'knows' everything e knows about the current evolution of the multi-threaded system except for the event e itself. Note that e may know less than Σ knows with respect to the evolution of other threads in the system, because Σ has global information.

The procedure *createState*(Σ, *msg*), which implements the function δ described in Corollary 19, creates a new global state Σ', where Σ' is a possible consistent global state that can result from Σ after the relevant event e that generated the message *msg*. Together with each state Σ in the lattice, a set of states of the monitor, *MonStates(*Σ*)*, also needs to be maintained, which keeps all the states of the monitor in which any of the partial runs ending in Σ can lead to. In the procedure *createState*, we set the *MonStates* of Σ' with the set of monitor states to which any of the current states in *MonStates*(Σ) can transit when the state Σ' is observed. *pgmState*(Σ') returns the value of all relevant program shared variables in state Σ', and *pgmState*(Σ')$[x \mapsto v]$ means that in *pgmState*(Σ') the relevant variable x is updated with the value v. Lemma 21 justifies that RVC of the state Σ' is updated properly.

The merging operation *nextLevel*$\uplus\Sigma$ adds the global state Σ to the set *nextLevel*. If Σ is already present in *nextLevel*, it updates the existing state's *MonStates* with the union of the existing state's *MonStates* and the *Monstates* of Σ. Two global states are same if their RVCs are equal.

The procedure *removeUselessMessages(CurrLevel,Q)* removes from Q all the message that cannot contribute to the construction of any state at the next level. It creates a RVC VR_{min} whose each component is the minimum of the corresponding component of the RVCs of all the global states in the set *CurrLevel*. It then removes all the messages in Q whose RVCs are less than or equal to V_{min}. This procedure makes sure that we do not store any unnecessary message. The correctness of the procedure is given by the following lemma.

Lemma 26. *For a given relevant event e, if* $VR\{e\} \leq VR_{min}$*, then* $\forall\Sigma \in$ *CurrLevel, e is not enabled in* Σ*.*

Proof. If e is enabled in Σ, then by Lemma 22, $VR\{e\}(t) = VC(\Sigma) + 1$, where t is the thread that

generated e. This implies that if e is enabled in Σ, then $VR\{e\} \not\leq VR\{\Sigma\}$. Since $VR\{e\} \leq VR_{min}$ we have $\forall \Sigma \in CurrLevel,\ VR\{e\} \leq VR\{\Sigma\}$. Therefore, e is not enabled in Σ. □

The observer runs in a loop till Q is empty. In each iteration of the loop, the procedure *constructLevel* is called. The pseudo-code for the observer is given in Figure 6.4.

6.3.3 Causality Cone Heuristic

The number of states on a level in the computation lattice can be exponential in the length of the trace. Generating all the states in a level may not be feasible due to limited memory. However, note that some states in a level can be considered more likely to occur in a consistent run than others. For example, two independent events that can possibly permute may have a huge time difference. Permuting these two events would give a consistent run, but that run may not be likely to take place in a real execution of the multi-threaded program. So we can ignore such a permutation. We formalize this concept as *causality cone*, or *window*, and exploit it in restricting our attention to a small set of states in a given level.

As mentioned earlier, we assume that the messages are received in an order in which they are generated in the execution. Note that this ordering corresponds to the real execution of the program, and it respects the partial order associated with the computation. This execution will be taken as a reference to compute the most probable consistent runs of the system.

If we consider all the messages generated by executing the program as a finite sequence of messages, then a lattice formed by any prefix of this sequence is a sub-lattice of the computation lattice $\mathcal{L}$. This sub-lattice, say $\mathcal{L}'$, has the following property: if $\Sigma \in \mathcal{L}'$, then for any $\Sigma' \in \mathcal{L}$ if $\Sigma' \rightsquigarrow^* \Sigma$, then $\Sigma' \in \mathcal{L}'$. We can see this sub-lattice as a portion of the computation lattice $\mathcal{L}$ enclosed by a cone. The height of this cone is determined by the length of the prefix of messages. We call this *causality cone*. All the states in $\mathcal{L}$ that are outside this cone cannot be determined from the prefix of messages. Therefore, they are outside the causal scope of the sequence of events corresponding to the messages in the prefix. As we consider longer prefixes, this cone moves down.

If we compute a RVC V_{max}, whose each component is the maximum of the corresponding component of the RVCs of all the messages in the message sequence, then V_{max} represents the RVC of the global state appearing at the tip of the cone. The tip of the cone, by Corollary 24, traverses

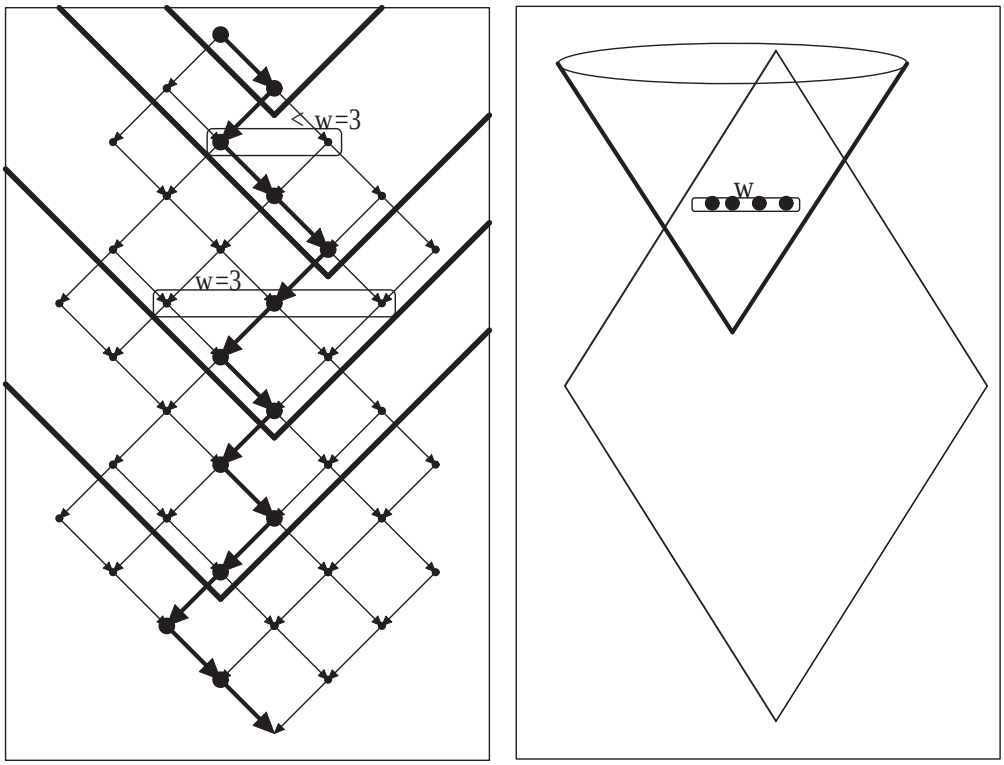

Figure 6.5: Causality Cones

```
levelComplete?(NextLevel, msg, Q)
    if size(NextLevel)≥ w
        return true;
    else if msg is the last message in Q
        return true;
    else
        return false;
```

Figure 6.6: *levelComplete?* Predicate

the actual execution run of the program.

To avoid the generation of a possibly exponential number of states in a given level, we consider a fixed number, say w, of most probable states in a given level. In a level construction, we say that the level is complete once we have generated w states in that level. However, a level may contain less than w states. Then the level construction algorithm gets stuck. To avoid this scenario, we say that a level is complete if we have used all the messages in the message sequence for the construction of the states in the current level . The pseudo-code for *levelComplete?* is given in Figure 6.6.

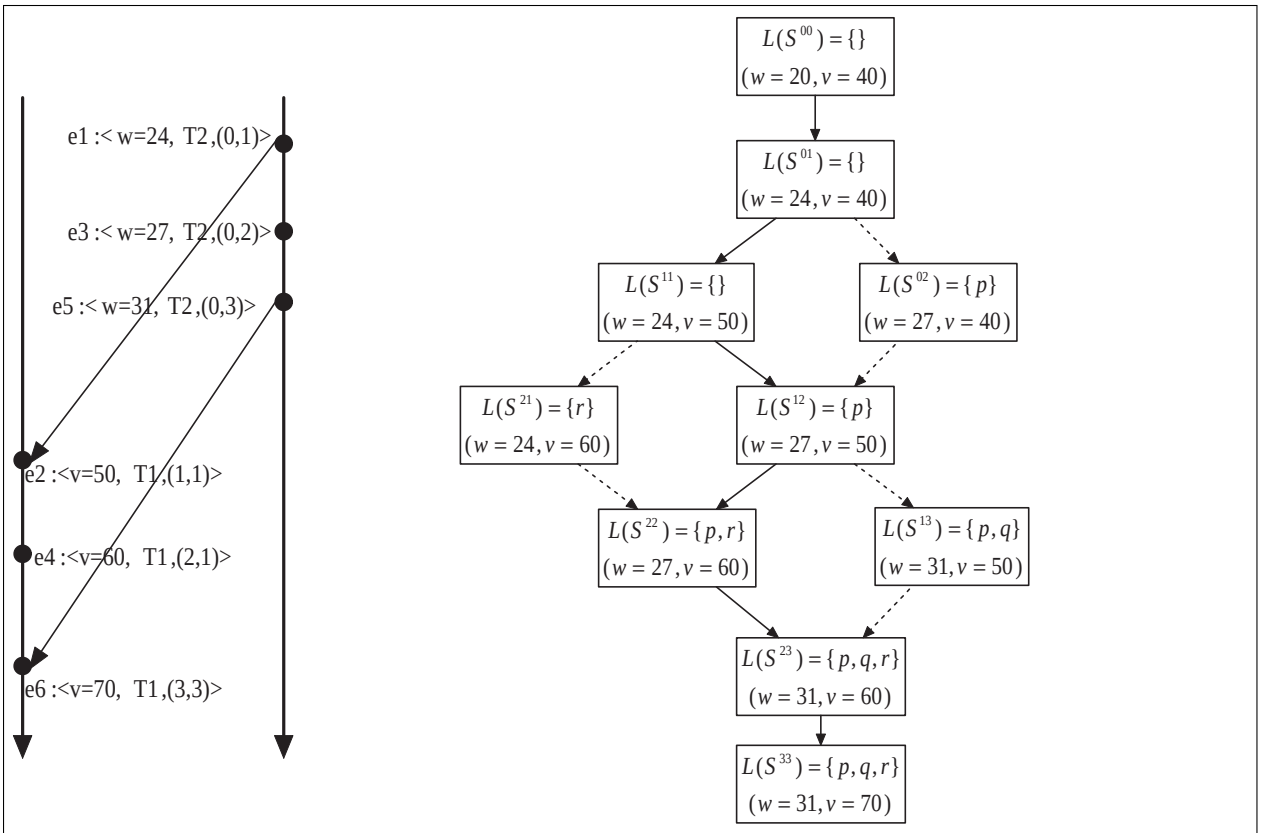

Figure 6.7: Causality Cone Heuristics applied to Example 2

Example 27. *Figure 6.7 shows the portion of the computation lattice constructed from the multi-threaded execution in Example 15, when the causality cone heuristics is applied with parameters* $w = 2$ *and* $l = 3$*. The possible consistent run* $\Sigma^{00}\Sigma^{01}\Sigma^{02}\Sigma^{03}\Sigma^{13}\Sigma^{23}\Sigma^{33}$*, shown on the left side of the Figure 6.7, is pruned out by the heuristics. In this particular run the two independent events* $e2$ *and* $e5$ *that are permuted have long time difference in the actual execution. Therefore, we can safely ignore this run among all other possible consistent runs.*

Chapter 7

Implementation and Case Studies

We first describe the details of two tools, which implement of the various testing methods described so far. We then report several case studies using the tools; these case studies show the applicability of the methods. In our case studies, we do not consider any formal specification. As such the case studies do not provide any empirical evaluation of the predictive monitoring technique.

7.1 Implementation

We have developed two automated concolic testing tools: CUTE for testing C programs and jCUTE for testing Java programs. CUTE only implements concolic testing and works for sequential C programs. jCUTE implements all three methods—concolic testing, race-detection and flipping, and predictive monitoring. The tools, CUTE and jCUTE, consist of two main modules: an instrumentation module and a library to perform symbolic execution, to solve constraints, to control thread schedules, and to perform predictive monitoring. The instrumentation module inserts code in the program under test so that the instrumented program calls the library at runtime for performing symbolic execution. The library creates a symbolic heap to symbolically track the shared memory locations. The library also maintains a symbolic stack for each thread to track the local variables symbolically. To solve arithmetic inequalities, the constraint solver of both CUTE and jCUTE uses `lp_solve` [61], a library for integer programming. jCUTE comes with a graphical user interface (a snapshot can be found in Figure 7.1).

7.1.1 Program Instrumentation

To instrument C code under test, CUTE uses CIL [68], a framework for parsing and transforming C programs. CUTE saves all the generated inputs in the file-system. Users of CUTE can replay

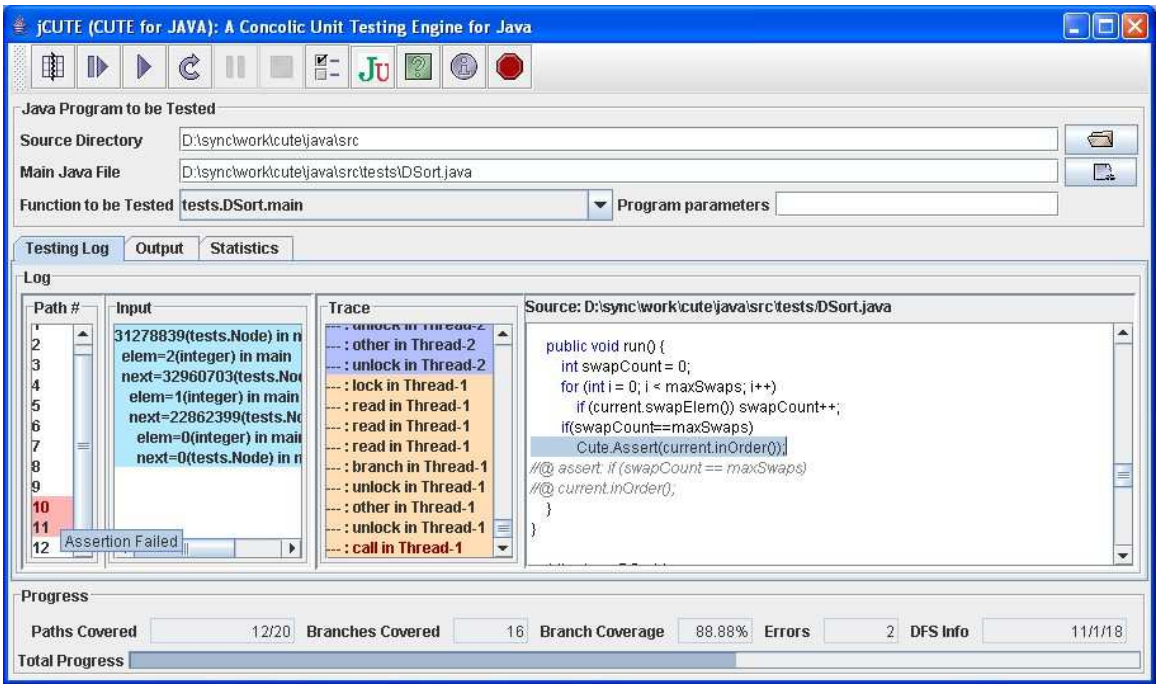

Figure 7.1: Snapshot of jCUTE

the program on these recorded inputs to reproduce the bugs. The replay can also be performed with the aid of a debugger. To instrument Java code under test, jCUTE uses the SOOT compiler framework [109]. jCUTE performs instrumentation at the bytecode level. For sequential programs, jCUTE can generate JUnit test cases, which can be used by the user for regression testing as well as for debugging. For concurrent programs, jCUTE records both the inputs and the schedules for various execution paths. This enables the user to replay the executions and reproduce bugs. jCUTE also allows the users to visualize graphically a multithreaded execution.

The instrumentation modules of both the tools first translate a program into the three-address code that closely follows the syntax given in Figure 3.1. The difference is that an expression e can also be a function call of the form $fun_name(v_1, \ldots, v_n)$. After the simplification, the instrumentation module inserts instrumentation code throughout the simplified code for concolic execution at runtime. Figure 7.1 shows examples of the code that the instrumentation module adds during instrumentation for function calls and function definitions. The procedure $push(\&v)$ pushes the symbolic expression for the address $\&v$ to a symbolic stack used for passing symbolic arguments during function calls. The reverse procedure $pop(\&v)$ pops a symbolic expression from the symbolic

Before Instrumentation	After Instrumentation
// function call $v = f(v_1, \ldots, v_n)$;	$push(\&v_1)$; ...; $push(\&v_n)$; $v = f(v_1, \ldots, v_n)$; $pop(\&v)$;

// function def
$Tf(T_1\ x_1, \ldots, T_n\ x_n)$ {
B; // body
`return v;` }
	$Tf(T_1\ x_1, \ldots, T_n\ x_n)$ {
	$pop(\&x_1)$; ...; $pop(\&x_n)$;
	B;
	$push(\&v)$;
	`return v;` }

Table 7.1: Code that the Instrumentation Module Adds for Functions.

stack and assigns it to the address $\&v$.

A difference between CUTE or jCUTE and traditional symbolic execution is that CUTE or jCUTE does not require instrumentation of the whole program. Calls to uninstrumented functions proceed only with the concrete execution, without symoblic execution. This allows CUTE or jCUTE to handle programs that use binary and native libraries whose source code or bytecode are not available.

7.1.2 Utility Functions

The CUTE toolkit provides two commands, `cutec` and `cute`, for code instrumentation and running of the instrumented code. The toolkit also provides four macros that give the user additional control over the instrumentation.

The command `cutec` expects a set of C files and a *toplevel* function; `cutec` instruments the C files and compiles the instrumented files with a C compiler. `cutec` assumes that the program starts by calling the toplevel function and that the input to the program consists of the memory graph reachable from the arguments passed to the toplevel function. `cutec` generates a `main` function that first initializes the input for the toplevel function and the symbolic state, and then calls the instrumented toplevel function with the generated input. At the end of the execution of the toplevel function, `main` calls the constraint solver to generate input for the next execution and stores the input in a file.

The command `cute` takes the executable generated by `cutec` and executes it iteratively until an error is found or *full branch coverage* is attained or a depth-first search completes. If an error

is found, `cute` invokes a debugger for the user to replay the erroneous execution.

The CUTE library provides the following macros that the user can insert into the C code under test:

1) `CUTE_input(x)` allows the user to specify that the variable `x` (of any type, including a pointer) is an input, besides the arguments of the toplevel function. This comes handy to replace any external user input, e.g., `scanf(``%d'',&v)` by `CUTE_input(v)` (which also assigns value to `&v`).

2) `CUTE_input_array(p,size)`. This macro is similar to `CUTE_input` except that it assumes that `p` is a pointer and specifies that `p` points to an array of size `size`.

3) `CUTE_assume(pred)`, where `pred` is some C predicate. This macro allows the execution to proceed if the `pred` holds. This way we can restrict the input, e.g., the predicate can be a `repOk()` call for some data structure.

4) `CUTE_assert(pred)`. This macro specifies an assertion whose violation is considered an error.

Similar commands and functions are also provided by jCUTE.

7.2 Experimental Evaluation

We illustrate six case studies, which show how CUTE and jCUTE can detect errors. In the first two case studies, we use CUTE for testing. In the rest of the case studies, we use jCUTE. The tool and the code for each case study can be found at `http://osl.cs.uiuc.edu/~ksen/cute/`. We ran the first two case studies on a Linux machine with a dual 1.7 GHz Intel Xeon processor. The rest of the case studies were run on a 2.0 GHz Pentium M processor laptop with 1 GB RAM running Windows XP.

7.2.1 Data Structures of CUTE

We applied CUTE to test its own data structures. CUTE uses a number of non-standard data structures at runtime, such as `cu_linear` to represent linear expressions, `cu_pointer` to represent pointer expressions, `cu_depend` to represent dependency graphs for path constraints etc. Our goal in this case study was to detect memory leaks in addition to standard errors such as segmentation faults, assertion violation etc. To that end, we used CUTE in conjunction with valgrind [108]. We discovered a few memory leaks and a couple of segmentation faults that did not show up in other

uses of CUTE. This case study is interesting in that we applied CUTE to partly unit test itself and discovered bugs. We briefly describe our experience with testing the `cu_linear` data structure.

We tested the `cu_linear` module of CUTE in the depth-first search mode of CUTE along with valgrind. In 537 iterations, CUTE found a memory leak. The following is a snippet of the function `cu_linear_add` relevant for the memory leak:

```
cu_linear *
cu_linear_add(cu_linear *c1, cu_linear *c2, int add) {
  int i, j;
  cu_linear* ret=(cu_linear*)malloc(sizeof(cu_linear));
  ... // skipped 18 lines of code
  if(ret->count==0) return NULL;
```

If the sum of the two linear expressions passed as arguments becomes constant, the function returns `NULL` without freeing the memory allocated for the local variable `ret`. CUTE constructed this scenario automatically at the time of testing. Specifically, CUTE constructed the sequence of function calls `l1=cu_linear_create(0); l1=cu_linear_create(0); l1=cu_linear_negate(l1); l1=cu_linear_add(l1,l2,1);` that exposes the memory leak that valgrind detects.

7.2.2 SGLIB Library

We also applied CUTE to unit test *SGLIB* [102] version 1.0.1, a popular, open-source C library for generic data structures. The library has been extensively used to implement the commercial tool Xrefactory. SGLIB consists of a single C header file, `sglib.h`, with about 2000 lines of code consisting only of C macros. This file provides generic implementation of most common algorithms for arrays, lists, sorted lists, doubly linked lists, hash tables, and red-black trees. Using the SGLIB macros, a user can declare and define various operations on data structures of parametric types.

The library and its sample examples provide verifier functions (can be used as `repOk`) for each data structure except for hash tables. We used these verifier functions to test the library using

the technique of `repOk` mentioned in Section 4.2. For hash tables, we invoked a sequence of its function. We used CUTE with bounded depth-first search strategy with bound 50. Figure 7.2 shows the results of our experiments.

We chose SGLIB as a case study primarily to measure the efficiency of CUTE. As SGLIB is widely used, we did not expect to find bugs. Much to our surprise, we found *two bugs* in SGLIB using CUTE.

The first bug is a segmentation fault that occurs in the doubly-linked-list library when a non-zero length list is concatenated with another zero-length list. CUTE discovered the bug in 140 iterations (about 1 seconds) in the bounded depth-first search mode. This bug is easy to fix by putting a check on the length of the second list in the concatenation function.

The second bug, which is a more serious one, was found by CUTE in the hash table library in 193 iterations (in 1 second). Specifically, CUTE constructed the following valid sequence of function calls which gets the library into *an infinite loop*:

```
typedef struct ilist { int i; struct ilist *next; } ilist;
ilist *htab[10];
main() {
  struct ilist *e,*e1,*e2,*m;
  sglib_hashed_ilist_init(htab);
  e=(ilist *)malloc(sizeof(ilist)); e->next = 0; e->i=0;
  sglib_hashed_ilist_add_if_not_member(htab,e,&m);
  sglib_hashed_ilist_add(htab,e);
  e2=(ilist *)malloc(sizeof(ilist)); e2->next = 0; e2->i=0;
  sglib_hashed_ilist_is_member(htab,e2); }
```

where `ilist` is a `struct` representing an element of the hash table. We reported these bugs to the SGLIB developers, who confirmed that these are indeed bugs.

Figure 7.2 shows the results for testing SGLIB 1.0.1 with the bounded depth-first strategy. For each data structure and array sorting algorithm that SGLIB implements, we tabulate the time that CUTE took to test the data structure, the number of runs that CUTE made, the number of

Name	Run time in seconds	# of Iterations	# of Branches Explored	% Branch Coverage	# of Functions Tested	OPT 1 in %	OPT 2 & 3 in %	# of Bugs
Array Quick Sort	2	732	43	97.73	2	67.80	49.13	0
Array Heap Sort	4	1764	36	100.00	2	71.10	46.38	0
Linked List	2	570	100	96.15	12	86.93	88.09	0
Sorted List	2	1020	110	96.49	11	88.86	80.85	0
Doubly Linked List	3	1317	224	99.12	17	86.95	79.38	1
Hash Table	1	193	46	85.19	8	97.01	52.94	1
Red Black Tree	2629	1,000,000	242	71.18	17	89.65	64.93	0

Table 7.2: Results for Testing SGLIB 1.0.1 with Bounded Depth-First Strategy with Depth 50

branches it executed, branch coverage obtained, the number of functions executed, the benefit of optimizations, and the number of bugs found.

The branch coverage in most cases is less than 100%. After investigating the reason for this, we found that the code contains a number of assert statements that were never violated and a number of predicates that are redundant and can be removed from the conditionals.

The last two columns in Figure 7.2 show the benefit of the three optimizations from Section 4.1.5. The column OPT 1 gives the average percentage of executions in which the fast unsatisfiability check was successful. It is important to note that the saving in the number of satisfiability checks translates into an even higher relative saving in the satisfiability-checking time because `lp_solve` takes much more time (exponential in number of constraints) to determine that a set of constraints is unsatisfiable than to generate a solution when one exists. For example, for red-black trees and depth-first search, OPT 1 was successful in almost 90% of executions, which means that OPT 1 reduces the number of calls to `lp_solve` an order of magnitude. However, OPT 1 reduces the solving time of `lp_solve` more than two orders of magnitude in this case; in other words, it would be infeasible to run CUTE without OPT 1. The column OPT 2 & 3 gives the average percentage of constraints that CUTE eliminated in each execution due to common sub-expression elimination and incremental solving optimizations. Yet again, this reduction in the size of constraint set translates into a much higher relative reduction in the solving time.

7.2.3 Java 1.4 Collection Library

We used jCUTE to test the thread-safe Collection framework implemented as part of the `java.util` package of the standard Java library provided by Sun Microsystems. A number of data structures provided by the package `java.util` are claimed as thread-safe in the Java API documentation.

This implies that the library should provide the ability to safely manipulate multiple objects of these data structures simultaneously in multiple threads. No explicit locking of the objects should be required to safely manipulate the objects. More specifically, multiple invocation of methods on the objects of these data structures by multiple threads must be equivalent to a sequence of serial invocation of the same methods on the same objects by a single thread.

We chose this library as a case study primarily to evaluate the effectiveness of our jCUTE tool. As Sun Microsystems' Java is widely used, we did not expect to find potential bugs. Much to our surprise, we found several previously undocumented data races, deadlocks, uncaught exceptions, and an infinite loop in the library. Note that, although the number of potential bugs is high, these bugs are all caused by a couple of problematic design patterns used in the implementation.

Experimental Setup The `java.util` provides a set of classes implementing thread-safe Collection data structures. A few of them are ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, HashMap, TreeMap, etc. The Vector class is synchronized by implementation. For the other classes, one needs to call the static functions such as `Collections.synchronizedList`, `Collections.synchronizedSet`, etc., to get a synchronized or thread-safe object backed by a non-synchronized object of the class. To setup the testing process we wrote a multithreaded test driver for each such thread-safe class. The test driver starts by creating two empty objects of the class. The test driver also creates and starts a set of threads, where each thread executes a different method of either of the two objects concurrently. The invocation of the methods strictly follows the contract provided in the Java API documentation. We created two objects because some of the methods, such as `containsAll`, takes as an argument an object of the same type. For such methods, we call the method on one object and pass the other object as an argument. Note that more sophisticated test drivers can be written. A simplified skeleton of the test-driver that we used is given below:

```
public class MTListTest extends Thread {
 List s1,s2;
 public MTListTest(List s1, List s2) {
   this.s1 = s1;  this.s2 = s2; }
```

```
  public void run() {
    int c = Cute.input.Integer();
    Object o1 = (Object)Cute.input.Object("java.lang.Object");
    switch(c){
     case 0: s1.add(o1); break;
     case 1: s1.addAll(s2); break;
     case 2: s1.clear(); break;
     .
     .} }
  public static void main(String[] args) {
    List s1 = Collections.synchronizedList(new LinkedList());
    List s2 = Collections.synchronizedList(new LinkedList());
    (new MTListTest(s1,s2)).start();
    (new MTListTest(s2,s1)).start();
    (new MTListTest(s1,s2)).start();
    (new MTListTest(s2,s1)).start();}
}
```

The arguments to the different methods are provided as input to the program. If a class is thread-safe, then there should be no error if the test-driver is executed with any possible interleaving of the threads and any input. However, jCUTE discovered data races, deadlocks, uncaught exceptions, and an infinite loop in these classes. Note that in each case jCUTE found no such error if methods are invoked in a single thread. As such the bugs detected in the Java Collection library are *concurrency related*.

The summary of the results is given in the Table 7.3. Here we briefly describe an infinite loop and a data race leading to an exception that jCUTE discovered in the synchronized `LinkedList` class and the synchronized `ArrayList` class, respectively.

We present a simple scenario under which the infinite loop happens. The test driver first creates two synchronized linked lists by calling

```
List l1 = Collections.synchronizedList(new LinkedList());
List l2 = Collections.synchronizedList(new LinkedList());
l1.add(null);
l2.add(null);
```

The test driver then concurrently allows a new thread to invoke `l1.clear()` and another new thread to invoke `l2.containsAll(l1)`. jCUTE discovered an interleaving of the two threads that resulted in an infinite loop. However, the program never goes into infinite loop if the methods are invoked in any order by a single thread. jCUTE also provided a trace of the buggy execution. This helped us to detect the cause of the bug. The cause of the bug is as follows. The method `containsAll` holds the lock on `l2` throughout its execution. However, it acquires the lock on `l1` whenever it calls a method of `l1`. The method `clear` always holds the lock on `l1`. In the trace, we found that the first thread executes the statements

```
modCount++;
header.next = header.previous = header;
```

of the method `l1.clear()` and then there is a context switch before the execution of the statement `size=0;` by the first thread. The other thread starts executing the method `containsAll` by initializing an iterator on `l1` without holding a lock on `l1`. Since the field `size` of `l1` is not set to 0, the iterator assumes that `l1` still has one element. The iterator consumes the element and increments the field `nextIndex` to 1. Then a context switch occurs and the first thread sets `size` of `l1` to 0 and completes its execution. Then the other thread starts looping over the iterator. In each iteration `nextIndex` is incremented. The iteration continues if the method `hasNext` of the iterator returns true. Unfortunately, the method `hasNext` performs the check `nextIndex != size;` rather than checking `nextIndex < size;`. Since `size` is 0 and `nextIndex` is greater than 0, `hasNext` always returns true and hence the loop never terminates. The bug can be avoided if `containsAll` holds lock on both `l1` and `l2` throughout its execution. It can also be avoided if `containsAll` uses the synchronized method `toArray` as in the `Vector` class, rather than using iterators. Moreover,

Name	Run time in seconds	# of Paths	# of Threads	% Branch Coverage	# of Functions Tested	# of Bugs Found data races/deadlocks/ infinite loops/exceptions
Vector	5519	20000	5	76.38	16	1/9/0/2
ArrayList	6811	20000	5	75	16	3/9/0/3
LinkedList	4401	11523	5	82.05	15	3/3/1/1
LinkedHashSet	7303	20000	5	67.39	20	3/9/0/2
TreeSet	7333	20000	5	54.93	26	4/9/0/2
HashSet	7449	20000	5	69.56	20	19/9/0/2

Table 7.3: Results for Testing Synchronized Collection Classes of JDK 1.4

the statement `nextIndex != size;` should be changed to `nextIndex < size;` in the method `hasNext`. Note that this infinite loop should not be confused with the infinite loop in the following wrongly coded sequential program commonly found in the literature.

```
List l = new LinkedList();  l.add(l); System.out.println(l);
```

We next present a simple scenario under which jCUTE found a data race leading to an uncaught exception in the class `ArrayList`. The test driver first creates two synchronized array lists by calling

```
List l1 = Collections.synchronizedList(new ArrayList());
List l2 = Collections.synchronizedList(new ArrayList());
l1.add(new Object());
l2.add(new Object());
```

The test driver then concurrently allows a new thread to invoke `l1.add(new Object())` and another new thread to invoke `l2.containsAll(l1)`. During testing, jCUTE discovered data races over the fields `size` and `modCount` of the class `ArrayList`. In a subsequent execution, jCUTE permuted the events involved in a data race and discovered an uncaught `ConcurrentModificationException` exception. However, the program never throws the `ConcurrentModificationException` exception if the methods are invoked in any order by a single thread. Note that the Java API documentation claims that there should be no such data race or uncaught `ConcurrentModificationException` exception when we use synchronized form of array list. jCUTE also provided a trace of the buggy execution. This helped us to detect the cause of the bug. It is worth mentioning that jCUTE not only detects actual races, but also flips to see if the data race can be fatal, i.e., that it can lead to uncaught exceptions.

7.2.4 NASA's Java Pathfinder's Case Studies

In [75], several case studies have been carried out using Java PathFinder and Bandera. These case studies involve several small to medium-sized multithreaded Java programs; thus they provide a good suite to evaluate jCUTE. The programs include RemoteAgent, a Java version of a component of an embedded spacecraft-control application, Pipeline, a framework for implementing multi-threaded staged calculations, RWVSN, Doug Lea's framework for reader writer synchronization, DEOS, a Java version of the scheduler from a real-time executive for avionics systems, BoundedBuffer, a Java implementation of multithreaded bounded buffer, NestedMonitor, a semaphore based implementation of bounded buffer, and ReplicatedWorkers, a parameterizable job scheduler. Details about these programs can be found in [75]. We also considered a distributed sorting implementation used in [53]. This implementation involves both concurrency and complex data inputs.

We used jCUTE to test these programs. Since most of these programs are designed to run in an infinite loop, we bounded our search to a finite depth. jCUTE discovered known concurrency related errors in RemoteAgent, DEOS, BoundedBuffer, NestedMonitor, and the distributed sorting implementation and seeded bugs in Pipeline, RWVSN, and ReplicatedWorkers. The summary of the results is given in the Table 7.4. In each case, we stopped at the first error. Note the although the running time of our experiments is many times smaller than that in [75, 53], we are also using a much faster machine.

It is worth mentioning that we tested the *un-abstracted version* of these programs rather than requiring a programmer to manually provide abstract interpretations as in [75]. This is possible with jCUTE because jCUTE tries to explore distinct paths of a program rather than exploring distinct states. Obviously, this means that we cannot prove a program correct if the program has infinite length paths. Java PathFinder and Bandera can verify a program in such cases if the state space of the abstracted program is finite.

7.2.5 Needham-Schroeder Protocol

The Needham-Schroeder public-key authentication protocol [77] aims at providing mutual authentication through message exchanges between two parties: an *initiator* and a *responder*; details of

Name	Run time in seconds	# of Paths	# of Threads	% Branch Coverage	Lines of Code	# of Bugs Found data races/deadlocks/assertions/exceptions
BoundedBuffer	11.41	43	9	100.0	127	0/1/0/0
NestedMonitor	0.46	2	3	100.0	214	0/1/0/0
Pipeline	0.70	3	5	64.29	103	1/0/1/0
RemoteAgent	0.45	2	3	87.5	55	1/1/0/0
RWVSN	2.19	8	5	68.18	590	1/0/1/0
ReplicatedWorkers	0.34	1	5	25.93	954	0/0/1/0
DEOS	35.23	111	6	64.75	1443	0/0/1/0

Table 7.4: Java PathFinder's Case Studies (un-abstracted)

the protocol can be found elsewhere [77]. Lowe reported an attack against the original protocol and also proposed a fix, called Lowe's fix [36].

We tested a concurrent implementation of the protocol using jCUTE. jCUTE found the attack in 406 iterations or about 95 seconds of search.

We compare these results with the ones reported previously [40, 41] for the same protocol. The explicit-state C model-checker VeriSoft [40] analyzed a concurrent implementation of the protocol with finite input domain. Verisoft was unable to find the attack within 8 hours, evolutionary testing (with manual tuning) found the attack after 50 minutes (on a somewhat slower machine). DART [41] found the attack on a sequential implementation of the protocol with a somewhat stronger intruder model[1] in 18 minutes. In comparison, jCUTE found the attack on a concurrent implementation of the protocol with a proper intruder model in only 95 seconds, which is an order of magnitude faster than the fastest previous approach. This performance difference is due to jCUTE's efficient algorithm that only explores distinct causal structures.

7.2.6 TMN Protocol

The Tatebayashi, Matsuzaki, and Newman (TMN) Protocol [107] is a protocol for distribution of a fresh symmetric key. In this protocol when an initiator wants to communicate with a responder, it uses a trusted server to obtain a secret symmetric session key. The details of the protocol can be found in [107].

In this protocol, an intruder can establish a parallel session through eavesdropping and obtain the secret key [60]. We tested a concurrent implementation of the protocol using jCUTE. jCUTE found the attack in 522 iterations or about 127 seconds of search.

[1]Note that a stronger intruder model makes it easier for the intruder to find an attack. This in turn makes the search space smaller resulting in faster testing time.

Chapter 8

Related Work

We briefly describe the related work and compare it to the work presented in this dissertation. The body of related work may be loosely classified into three broad categories: testing sequential programs, testing concurrent programs, and runtime verification.

8.1 Testing Sequential Programs

Automated testing is an active area of research. In the last five years, over a dozen of techniques and tools have been proposed that automatically increase test coverage or generate test inputs.

The simplest, and yet often very effective, techniques use random generation of (concrete) test inputs [15, 70, 35, 24, 72]. Some recent tools use bounded-exhaustive concrete execution [117, 42, 16]; such testing tries all values from user-provided domains. These tools can achieve high code coverage, especially for testing data structure implementation. However, they require the user to carefully choose the values in the domains to ensure the high coverage.

Tools based on symbolic execution use a variety of approaches; in our view, the most relevant of these are abstraction-based model checking [10, 14], explicit-state model checking [115], symbolic-sequence exploration [118, 73], and static analysis [25]—to detect (potential) bugs or generate test inputs. These tools inherit the incompleteness of their underlying reasoning engines such as theorem provers and constraint solvers. For example, tools using precise symbolic execution [115, 118] cannot analyze any code that would build constraints out of pre-specified theories, e.g., any code with non-linear arithmetic or array indexing with non-constant expressions. As another example, tools based on predicate abstraction [10, 14] do not handle code that depends on complex data structures. In these tools, the symbolic execution proceeds separately from the concrete execution (or constraint solving).

Cadar and Engler propose *Execution Generated Testing* (EGT) [18], an approach similar to CUTE: EGT explores different execution paths using a combined symbolic and concrete execution. However, EGT does not consider inputs that are memory graphs or code that has preconditions. Also, EGT and CUTE differ in how they approximate symbolic expressions with concrete values. EGT follows a more traditional approach to symbolic execution and proposes an interesting method that *lazily* solves the path constraints: EGT starts with only symbolic inputs and tries to execute the code fully symbolically, but if it cannot do so, EGT solves the current constraints to generate a (partial) concrete input with which the execution proceeds.

CUTE is also related to some methods which use backtracking to generate a test input that executes one given path (e.g., a path that may be known to contain a bug) [56, 46]. However, in contrast to these methods, CUTE attempts to cover *all* feasible paths, in a style similar to systematic testing. Moreover, the prior work did not address inputs that are memory graphs. Visvanathan and Gupta [116] proposed a technique that generates memory graphs. They also use a specialized symbolic execution (not the exact execution with symbolic arrays) and develop a solver for their constraints. However, they consider one given path, do not consider unknown code segments (e.g., library functions), and do not use a combined concrete execution to generate new test inputs. Moreover, in our case the constraint solving is incremental.

8.2 Testing Concurrent Programs

Improving the reliability of concurrent programs is a challenging area of research. A major cause for defects in multithreaded programs is race conditions. A large body of research focuses on dynamic or static race detection [78, 69, 33, 82, 21, 26]. Race detection suffers from the problem of false warnings. Moreover, the dynamic techniques can report all possible race conditions only if there are good test inputs that can achieve high code coverage. Our algorithm not only detects races but also permutes them systematically to search if the races can lead to some bug. Moreover, jCUTE generates test inputs so that the number of races caught is maximized.

Bruening [17] first proposed a technique for *dynamic partial order reduction*, called ExitBlock-RW algorithm, to systematically test multithreaded programs. These technique uses two sets, *delayed set* and *enabled set*, similar to the sets *postponed* and T_{enabled} in our algorithm, to enumerate

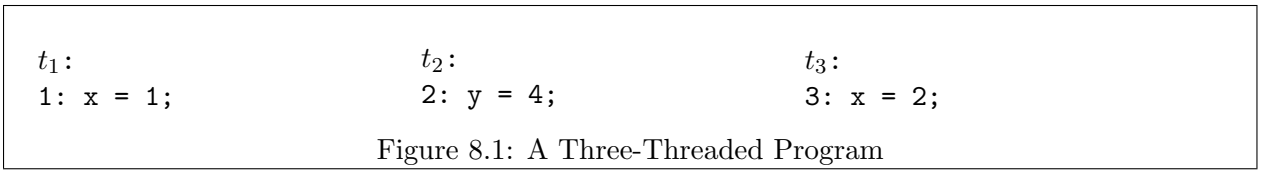

Figure 8.1: A Three-Threaded Program

meaningful schedules by re-ordering dependent atomic blocks. However, this assume that the program under test follows a consistent mutual-exclusion discipline using locks. The dynamic partial order reduction technique proposed by Carver and Lei [19] guarantees that exactly one interleaving for each partial order is explored. However, their approach involves storing schedules that have not been yet explored; this can become a memory bottleneck.

More recently, dynamic partial order reduction proposed by Flanagan and Godefroid [34] removes the memory bottleneck in [19] at the cost of possibly exploring more than one interleaving for each partial order. This technique uses dynamically constructed *persistent sets* and *sleep sets* [38] to prune the search space. The key difference between the DPOR algorithm in [34] and our race-detection and flipping algorithm is that,for every choice point, the DPOR algorithm in [34] uses a persistent set and we use a postponed set. These two sets can be different at a choice point. For example, for the 3-threaded program in Figure 8.1, if the first execution path is $(t_1, 1, \mathbf{w})(t_2, 2, \mathbf{w})(t_3, 3, \mathbf{w})$, then at the first choice point denoting the initial state of the program, the persistent set is $\{t_1, t_3\}$; whereas, at the same choice point, the postponed set is $\{t_1\}$. (Apart from scheduling the thread t_1, the race-detection and flipping algorithm also schedules the thread t_2 at the first choice point.) Note that the DPOR algorithm in [34] picks the elements of a persistent set by using a complex forward lookup algorithm. In contrast, we *simply* put the current scheduled thread to the postponed set at a choice point.

Moreover, the implementation in [34] considers two read accesses to the same memory location by different threads to be dependent. Thus for the 3-threaded program in Figure 8.2, the implementation described in [34] would explore six interleavings. We remove the redundancy associated with this assumption by using a more general notion of race and its detection using dynamic vector clock algorithm. As such, for the above example, we will explore only one interleaving. Note that none of the previous descriptions of the above dynamic partial order reduction techniques have handled programs which have inputs.

In a similar independent work [103], Siegel et al. use a combination of symbolic execution and

t_1:	t_2:	t_3:
1: lv1 = x;	2: lv2 = x;	3: if (x > 0) 4: ERROR;

Figure 8.2: Another Three-Threaded Program

static partial order reduction to check if a parallel numerical program is equivalent to a simpler sequential version of the program. Thus this work can also be seen as a way of combining symbolic execution with partial order reduction based model checking techniques for the purpose of testing parallel programs. However, their work deals with symbolic execution of numerical programs with floating points, rather than programs with pointers and data-structures. Therefore, static partial order reduction proves effective in their approach.

Model checking tools [106, 23, 29] based on static analysis have been developed. These tools are useful in detecting bugs in concurrent programs. These tools also employ (static) partial order reduction techniques to reduce search space. The partial order reduction depends on detection of thread-local memory locations and patterns of lock acquisition and release. Because of the use of static analysis, the methods can result in false warnings. However, despite encouraging recent successes in software model checking for larger systems, there is little hope that one can actually prove the correctness of large concurrent systems, and one must in those cases still rely on debugging and testing.

8.3 Runtime Verification

There has been considerable interest in runtime verification techniques in recent years, as perhaps best shown by the series of workshops [1]. Runtime verification can be simplistically viewed as a rigorous approach to testing, in which the requirements specifications use some underlying logical, typically temporal [62, 63], formalism. The same algorithms used to detect errors during testing can be used to trigger recovery actions at runtime, so runtime verification techniques are frequently applied in monitoring. Runtime verification has so far been concerned with analyzing software systems, essentially as a complementary approach to model checking software systems [11, 39, 48, 114, 52, 23, 74, 106]. We briefly review some of the techniques for runtime verification below. Note that these techniques may also possibly be combined with our method for predictive monitoring.

NASA's runtime verification system Java PathExplorer (JPaX) [49] and its sub-system EAGLE [13] has already been used to analyze the K9 Mars Rover [7]. EAGLE has also been used to find security attacks in DARPA logs [67]. Temporal Rover and DBRover [27, 28] are commercial runtime verification tools. The MaC tool [59, 54] is a runtime monitoring tool with a specialized formal monitoring specification language with the potential for steering the execution of programs at runtime. A technique is proposed in [57] where the execution events are stored in an SQL database at runtime and then analyzed by means of queries after the program terminates. The PET tool, described in [45, 44, 43], uses a future time temporal logic formula to guide the execution of a program for debugging purposes. POTA and Java MultiPathExplorer [83, 90, 92, 97] are tools which check safety formulae against a partial order extracted online from an execution trace. Efficient decentralized monitoring of message passing distributed systems is proposed in [95, 96]. Java-MoP [20] proposes the use of monitoring as a programming paradigm. Complexity results for testing a finite trace against temporal formulae expressed in different logics are investigated in [64]. Algorithms using alternating automata to monitor temporal properties are proposed in [32], and a specialized collecting statistics technique along the execution trace is described in [31]. Various algorithms to generate testing automata from temporal logic formulae are discussed in [79, 71], and [37] presents a Büchi automata inspired algorithm.

Chapter 9

Conclusion

With the increasing use of software in society, we want software to be reliable, safe, secure, and robust. Unfortunately, as experience in industry has shown, developing large reliable software is a hard task. As such more than half of the total software development cost is spent in testing. Even after such a huge investment, serious bugs and security flaws are common in widely-used software. We believe this is because the most widely used method for improving software quality—testing—is generally done manually and in *ad hoc* ways. Automated and rigorous methods are rarely used for testing because such methods are hardly effective or scalable. In this dissertation, we developed a family of methods of automated testing, which we believe can improve testing and make it more systematic and scalable. Our methods use ideas from formal methods, constraint solving, theory of concurrency, dynamic program analysis, and model checking and apply them to build effective testing methods which exploit the full computational power of modern day computers in the testing process. The net result is that we are able to make testing systematic and automated. We summarize the contributions of this dissertation in the next section, and discuss issues and open problems in the final section.

9.1 Summary

We presented concolic testing, a method to explore different paths in programs by coupling concrete and symbolic executions in a cooperative way. Concrete execution enables symbolic execution to mitigate the effects of the incompleteness of the underlying reasoning engines; for example, concrete execution helps resolve the constraints that theorem provers cannot handle, resolve aliases for pointers (using concrete values for pointers), handle arrays and pointers (our technique requires no static alias analysis) etc. On the other hand, symbolic execution (along with constraint solving at

the end of execution) helps generate concrete inputs that lead the program to a different concrete execution, thus increasing coverage. We showed that constraint solving for concolic testing can be done in an incremental way, which makes test input generation highly efficient. We described how to efficiently generate dynamic data structures by incrementally adding or removing a node, or by aliasing two pointers.

Concolic testing works only for sequential programs. Testing becomes notoriously hard for large concurrent software due to the inherent non-determinism in the execution of such software. We extended the concolic testing approach to develop a method for testing concurrent programs. The extended method uses the concrete execution of concolic testing to determine an abstract relation, called *causality relation*, between the events in a concurrent execution. This causality relation naturally defines an equivalence relation between the execution paths of a concurrent program. We provide a technique for exploring at least one candidate from each equivalence class of execution paths of a concurrent program with complex data inputs; this improves the efficiency of testing concurrent programs considerably. In addition to common errors such as assertion violations, memory leaks, uncaught exceptions, and segmentation faults, our testing approach can catch concurrency related errors such as data races and deadlocks. Because our testing approach is designed to explore execution paths of a concurrent program, we term our approach *Explicit Path Model Checking*.

In other research [84], we have developed a method that extend concolic testing to test message passing distributed systems or actor systems [3, 5]. In this method, we assume that a program consists of a number of asynchronously executing concurrent processes or actors, which may take data inputs and communicate using asynchronous messages. Because of the large numbers of possible data inputs as well as the asynchrony in the execution and communication, distributed programs exhibit very large numbers of potential behaviors. As in the race-detection and flipping algorithm, our method uses simultaneous concrete and symbolic execution, or *concolic execution*, to explore all distinct behaviors that may result from a program's execution given different data inputs and schedules. The key idea is as follows. We use the symbolic execution to generate data inputs that may lead to alternate behaviors. At the same time, we use the concrete execution to determine, at runtime, the partial order of events in the program's execution. This enables us to improve the efficiency of our algorithm by avoiding many tests which would result in equivalent

behaviors. We have implemented this method in jCUTE. In order to keep the dissertation concise and coherent, we did not include a description this method.

Shared memory systems can be modeled as asynchronous message passing systems by associating a thread with every memory location. Reads and writes of a memory location can be modeled as asynchronous messages to the thread associated with the memory location. However, this particular model would treat both reads and writes similarly. Hence, the algorithm in [84] would explore many redundant executions. For example, for the 2-threaded program $t_1 : x = 1; x = 2; t_2 : y = 3; x = 4;$, the algorithm in [84] would explore six interleavings. Our race-detection and flipping algorithm, which assumes that two reads are not in race, would explore only three interleavings of the program.

We observed that concolic testing can miss bugs if we test concurrent multi-threaded programs against a formal specification. In particular we showed that exploring only one candidate execution path from each equivalence class is *not sufficient* for catching violations of temporal properties; a temporal property may be simultaneously satisfied and violated by two different causally equivalent execution paths. To solve this problem, we proposed a testing method based on predictive monitoring of concurrent programs. In this technique, from an observed execution path, we statically generate all the causally equivalent execution paths and represent such paths compactly in an abstract model called computation lattice. We showed that monitoring of temporal properties can be done efficiently using this model. Using this technique, we can predict violations of properties in non-observed execution paths without re-executing the program; therefore, we call this technique *predictive monitoring*. It is important to note that, although predictive monitoring can predict and monitor all execution paths that are causally equivalent to an observed execution path, we still need concolic testing along with race-detection and flipping to explore all non-equivalent execution paths.

Based on these methods we have developed tools for testing C and Java programs. The binaries of these tool have been made available to researchers. These tools serve as a core engine to effectively explore non-equivalent execution paths of a programs. One application of this basic functionality is unit testing, which is provided as a feature in these tools. However, the possibilities of applying these tools are endless. The basic path exploration feature of these tools can also be used to generate regression test suites, to detect likely program invariants, to carryout simulations, for

stress testing, and as a part of other dynamic analysis methods.

9.2 Discussion

We presented a set of methods to effectively test shared-memory multi-threaded programs. Although the methods are quite effective in finding bugs in real-world programs, a number of questions are often raised about these methods. Next we try to address these questions. Our goal is to clarify the applicability and the limitations of our methods, as well as to describe open problems.

9.2.1 Scalability

A question that is often asked is how scalable our methods are. First, let us discuss the scalability of CUTE. Our experience shows that scalability in terms of memory usage is not a problem for CUTE because we explore one path at a time. During a concolic execution along a path, extra memory is consumed to maintain the symbolic state and the symbolic path constraint. It is often the case that a small fraction of the concrete state of a program is data dependent on the inputs. Therefore, the size of the symbolic state, which is almost proportional to the size of the part of the concrete state that is data dependent on the inputs, usually remains small. Moreover, along an execution path, the number of conditionals that are data dependent on the inputs are often small. This results in low memory usage for the maintenance of path constraint. Note that in the case of model checking, the tools tend to keep the entire state space in the memory and memory often becomes a bottleneck.

Obviously, the number of paths of a large program can be huge. Exploration of huge number of paths may take a lot of time. We call this the *path explosion problem*. If we put a time limit on the process of testing for a program having a huge number of paths, a depth-first search strategy often ends up in exploring a small subtree of the whole computation tree. As a result concolic testing gets *localized* to a small part of the computation tree. A way to address this problem would be to develop better search strategies. We describe a couple of candidate search strategies and discuss their pros and cons.

One possible search strategy is *random search* where a constraint to be solved is picked randomly from the set of constraints generated along a path. We believe that this strategy would tend

to sample the paths in the computation tree uniformly—thus, preventing the search from being localized to a small subtree. Obviously a limitation of the random search strategy is that concolic testing may end up exploring same path more than once. Moreover, unlike a depth-first search strategy, a random search strategy has no way to determine if it has explored the entire computation tree. Therefore, even for a small program, for which we can solve any generated path constraint, with random testing we have no way to prove that we have explored all the reachable statements of the program.

Another possible search strategy that prevents the search from getting localized is a *breadth-first search strategy*. Unfortunately, in the case of a breadth-first search strategy, for each depth of a tree, we have to store an input for each execution path passing through all the nodes at the depth in the tree. Since the number of nodes at a depth can be, in the worst case, equal to the number of the feasible execution paths of a program, the amount of storage required for storing the inputs for each path in a large programs may easily exhaust available persistent memory.

Developing interesting and efficient search strategies requires further investigation. Apart from investigating new efficient search strategies, one can also think of combining static analyses with concolic testing. Using static analyses, one can identify the subtrees in the whole computation tree that are problematic; one can then use concolic testing to explore only those subtrees. We believe that static analyses can help to prune the path space more aggressively.

Now let us turn to the scalability of jCUTE. In case of shared-memory multi-threaded programs that jCUTE deals with, a large number of accesses to the shared memory by various threads may result in a large number of non-equivalent execution paths. As a result jCUTE may not scale for such programs in principle. However, in practice, we observed that for 'well-written' programs (see [98]) an execution path often consists of large execution blocks that are *atomic*. The execution of such atomic blocks by a thread does not interfere with the execution of the other threads. A common way to ensure such atomicity is through the use of locks. During the process of testing such well-written programs, jCUTE ends up exploring the execution paths that are the interleavings of these large atomic blocks. As such jCUTE prunes a large portion of the path space. Obviously, such kind of pruning is not possible for programs in which multiple threads often access the shared memory without synchronization. Therefore, for the purpose of effective testing, one should avoid

such bad programming styles.

Apart from the above mentioned issues, the need for solving constraints may also prevent CUTE or jCUTE from scaling for large programs. When an execution path gets large, the size of the path constraint increases; eventually constraint solvers may not be able to handle the path constraint. However, constraint solving is very efficient in CUTE or jCUTE. This is because of the approximations that concolic testing performs: it separates the arithmetic constraints and the pointer constraints. If the last negated constraint is an arithmetic constraint, then only the set of arithmetic constraints is solved; otherwise, the set of pointer constraints is solved. Note that this way of decoupling the arithmetic constraints from the pointer constraints has some limitations (see the discussion in Section 4.1.6). However, we believe that such decoupling is a good compromise for the efficiency it provides.

Moreover, the solution in one execution path is often quite similar to the next execution as we negate only a single constraint. To exploit this fact, we proposed an optimization in Section 4.1.5. This optimization makes constraint solving incremental and highly efficient in case the negated constraint is an arithmetic constraint. In case the negated constraint is a pointer constraint, this becomes even more efficient as we simply add a node, delete a node, or make two pointers equal. Overall, we observed that incremental constraint solving takes a small fraction of the total execution time. Note that, although CUTE and jCUTE use a custom incremental solver, other solvers such as CVC Lite [12] or Uclid [58] can also be used for generating new inputs.

For shared-memory multi-threaded programs, our race-detection and flipping algorithm's efficiency depends on the number of equivalence classes of a given program—the larger the number of equivalence classes the lesser is the efficiency. A natural question to ask is whether the equivalence relation defined in Section 3.2 is optimal for efficiency. Unfortunately, the relation is not optimal. In [93], we showed that we can define a coarser equivalence relation that results in fewer number of equivalence classes. The equivalence relation uses a weak *happens-before relation* which orders a write of a shared variable with all its subsequent reads that occur before the next write to the variable. However, we do not know how to adapt this coarser equivalence relation in testing to obtain further reduction in the path space. This remains a topic for future research. Moreover, it would be useful to investigate the optimal equivalence relation for which we can develop a testing

method that would explore one candidate from each equivalence class.

9.2.2 Program Verification

A report of a bug by CUTE or jCUTE represents an actual bug because the bug is found by executing a program on a concrete input and schedule. However, CUTE and jCUTE can verify a program only in some limited cases: namely, if the following three conditions are satisfied. First, the testing process using CUTE (resp. jCUTE) terminates. Second, CUTE (resp. jCUTE) makes no approximation during concolic execution. Third, CUTE (resp. jCUTE) is able to solve any constraint which is satisfiable. These conditions guarantee that the tool has executed all feasible execution paths of a program and has hit all the reachable statements of the program. For most practical programs, CUTE and jCUTE can only find bugs; they cannot verify the programs. This can be seen as a limitation of our testing methods.

To partially address the above problem we have proposed two approaches: *statistical model checking* [99, 100, 101, 6, 4] and *learning to verify* [112, 111, 113]. Statistical model checking aims to bound the confidence with which we can say that a system is correct; in this approach, we check if a system whose behavior can be modeled probabilistically meets its formal reliability specification. Specifically, we assume that the specification is given in some probabilistic temporal logic such as the probabilistic computation tree logic (PCTL) [47] or the continuous stochastic logic (CSL) [2, 8]. Model checking is performed by automatically translating the specification into a series of inter-dependent statistical hypothesis testing experiments. The experiments are then conducted through discrete event simulation of the system. The number of simulation runs that need to be performed depends on the degree of confidence that we want in our decision.

Statistical model checking works for any system for which the inputs come from a fixed probability distribution. Unfortunately, for general software systems, one cannot assume there is a fixed probability distribution over the inputs. Therefore, we cannot directly apply statistical model checking to a general software system. We believe that by observing the actual behaviors exhibited by a system in the field, one can try to check if the inputs come from a fixed probability distribution; if this is the case, we can apply statistical model checking to verify quantitative properties of the system.

Another approach to verify infinite state systems is the *learning to verify* paradigm. The key idea behind this approach is based on the observation that often the state-space of infinite state systems is highly structured. For example, for a number of practical systems, such as parameterized systems, or systems with unbounded integers and message queues, the reachable state-space is regular and can be represented by a deterministic finite automaton. Traditional model checking techniques typically verify such a system by iteratively 'traversing' the entire state space of the system, but such traversal may take a long time to terminate, or it may never terminate. By using language inference and learning techniques, we have shown that it is often possible to find the reachable states of a system by either collecting its samples or by answering certain membership and equivalence queries. We have implemented this technique in a tool called LEVER and used it to analyze various systems with integers, message queues, stacks and parameterized systems. The systems that we analyzed using LEVER are quite small in size. A future research challenge would be make learning to verify work for large software systems where the reachable state space may not be regular—for example, by using concolic testing to obtain samples.

References

[1] *1st, 2nd, 3rd, 4th, and 5th Workshops on Runtime Verification (RV'01 - RV'05)*, volume 55(2), 70(4), 89(2), 113 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science: 2001, 2002, 2003, 2004, 2005, 2005.

[2] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time Markov chains. In *Proc. of Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 269–276, 1996.

[3] G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.

[4] G. Agha, C. Gunter, M. Greenwald, S. Khanna, J. Meseguer, K. Sen, and P. Thati. Formal modeling and analysis of DoS using probabilistic rewrite theories. In *International Workshop on Foundations of Computer Security (FCS'05) (Affiliated with LICS'05)*, 2005.

[5] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[6] G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. *Electronic Notes in Theoretical Computer Science*, 153(2):213–239, 2006. Expanded version of the paper that appeared in Proceedings of the 3rd Workshop on Quantitative Aspects of Programming Languages.

[7] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Roşu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2–3):209–234, May 2005.

[8] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous-time Markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[9] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[10] T. Ball. Abstraction-guided test generation: A case study. Technical Report MSR-TR-2003-86, Microsoft Research, 2003.

[11] T. Ball and S. Rajamani. The SLAM Toolkit. In *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, volume 2102 of *LNCS*, pages 260–264, 2001.

[12] C. W. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th International Conference on Computer Aided Verification*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.

[13] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.

[14] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Test from Counterexamples. In *Proc. of the 26th ICSE*, pages 326–335, 2004.

[15] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[16] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. of International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[17] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[18] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. of SPIN Workshop*, 2005.

[19] R. H. Carver and Y. Lei. A general model for reachability testing of concurrent programs. In *6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 76–98, 2004.

[20] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 106–125. Elsevier Science, 2003.

[21] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 258–269, 2002.

[22] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *Proceedings of the ESEC/FSE-9*, pages 142–151, 2001.

[23] J. Corbett, M. B. Dwyer, J. Hatcliff, C. S. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. of ICSE'00: International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.

[24] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[25] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *27th International Conference on Software Engineering*, 2005.

[26] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.

[27] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.

[28] D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proc. of CAV'03: Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 114–118. Springer-Verlag, 2003.

[29] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.

[30] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the Workshop on Parallel and Distributed Debugging (WPDD)*, pages 183–194. ACM, 1988.

[31] B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting Statistics over Runtime Executions. In *Proc. of RV'02: The Second International Workshop on Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2002. Elsevier.

[32] B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proc. of RV'01: The First International Workshop on Runtime Verification*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2001. Elsevier Science.

[33] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Proc. of the Program Analysis for Software Tools and Engineering Conference*, June 2001.

[34] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of the 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 110–121, 2005.

[35] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, 2000.

[36] G. Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Inf. Processing Letters*, 1995.

[37] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proc. of ASE'01: International Conference on Automated Software Engineering*, pages 412–416. Institute of Electrical and Electronics Engineers, 2001.

[38] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer-Verlag, 1996.

[39] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *24th ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[40] P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2002.

[41] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[42] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis*, pages 112–122, 2002.

[43] E. Gunter and D. Peled. Tracing the Executions of Concurrent Programs. In *Proc. of RV'02: Second International Workshop on Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[44] E. L. Gunter, R. P. Kurshan, and D. Peled. PET: An interactive software testing tool. In *Computer Aided Verification (CAV'00)*, volume 1885 of *Lecture Notes in Computer Science*, pages 552–556. Springer-Verlag, 2003.

[45] E. L. Gunter and D. Peled. Using functional languages in formal methods: The PET system. In *Parallel and Distributed Processing Techniques and Applications*, pages 2981–2986. CSREA, 2000.

[46] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proc. of the International Conference on Automated Software Engineering*, pages 219–227, 2000.

[47] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[48] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, Apr. 2000.

[49] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, to appear.

[50] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[51] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, January 2002.

[52] G. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*. IEEE/ACM, 1999.

[53] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th Int. Conf. on TACAS*, pages 553–568, 2003.

[54] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[55] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[56] B. Korel. A dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, November 1990.

[57] D. Kortenkamp, T. Milam, R. Simmons, and J. Fernandez. Collecting and Analyzing Data from Distributed Control Programs. In *Proc. of RV'01: First International Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*, Paris, France, 2001. Elsevier Science.

[58] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In *Proc. of Intl. Conf. on Computer-Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 475–478. Springer, 2004.

[59] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[60] G. Lowe and A. W. Roscoe. Using csp to detect errors in the TMN protocol. *Software Engg.*, 23(10):659–669, 1997.

[61] lp_solve. `http://groups.yahoo.com/group/lp_solve/`.

[62] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[63] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.

[64] N. Markey and P. Schnoebelen. Model checking a path (preliminary report). In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR'2003)*, Lecture Notes in Computer Science. Springer, 2003, to appear.

[65] K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WADG'91)*, volume 579 of *Lecture Notes in Computer Science*, pages 254–272. Springer-Verlag, 1991.

[66] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposia in Applied Mathematics*. AMS, 1967.

[67] P. Naldurg, K. Sen, and P. Thati. A temporal logic based approach to intrusion detection. In *24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'04)*, volume 3235 of *LNCS*, pages 359–376. Springer, 2004.

[68] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and transformation of C Programs. In *Proceedings of Conference on compiler Construction*, pages 213–228, 2002.

[69] R. Netzer and B. Miller. Detecting data races in parallel program executions. In *Advances in Languages and Compilers for Parallel Computing*. MIT Press, 1990.

[70] J. Offut and J. Hayes. A Semantic Model of Program Faults. In *Proc. of ISSTA'96*, pages 195–200, 1996.

[71] T. O'Malley, D. Richardson, and L. Dillon. Efficient Specification-Based Oracles for Critical Systems. In *Proc. of the California Software Symposium*, 1996.

[72] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *19th European Conference Object-Oriented Programming*, 2005.

[73] Parasoft. Jtest manuals version 6.0. Online manual, February 2005. `http://www.parasoft.com/`.

[74] D. Y. Park, U. Stern, and D. L. Dill. Java Model Checking. In *Proc. of the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000.

[75] C. S. Pasareanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *International Journal on Software Tools for Technology Transfer (STTT'03)*, 5(1):34–48, 2003.

[76] D. Peled. All from one, one for all: on model checking using representatives. In *5th Conference on Computer Aided Verification*, pages 409–423, 1993.

[77] R. Needham and M. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[78] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proc. of the SIGMETRICS symposium on Parallel and distributed tools*, pages 40–47, 1998.

[79] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *Proc. of ICSE'92: International Conference on Software Engineering*, pages 105–118, 1992.

[80] G. Roşu and K. Sen. An instrumentation technique for online analysis of multithreaded programs. In *Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD'04) (Satellite workshop of IPDPS'04)*. IEEE digital library, April 2004.

[81] G. Roşu and K. Sen. An instrumentation technique for online analysis of multithreaded programs. *Special Issue of Concurrency and Computation: Practice and Experience (CC:PE)*, 2006. (To Appear).

[82] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[83] A. Sen and V. K. .Garg. Partial order trace analyzer (pota) for distrubted programs. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, Electronic Notes in Theoretical Computer Science, 2003.

[84] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering (FASE'06)*, LNCS (To appear), 2006.

[85] K. Sen and G. Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical Report UIUCDCS-R-2006-2676, UIUC, 2006.

[86] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification (CAV'06)*, LNCS, 2006. (To Appear).

[87] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.

[88] K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. In *Proceedings of the 3rd Workshop on* Runtime Verification (RV'03), volume 89 of *ENTCS*, pages 162–181. Elsevier Science, 2003.

[89] K. Sen, G. Roşu, and G. Agha. Generating Optimal Linear Temporal Logic Monitors by Coinduction. In *Proceedings of 8th Asian Computing Science Conference (ASIAN'03)*, volume 2896 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, December 2003.

[90] K. Sen, G. Roşu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 337–346. ACM, 2003.

[91] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of 4th joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*. ACM, 2003.

[92] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138, Barcelona, Spain, March 2004.

[93] K. Sen, G. Roşu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Proceedings of 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *LNCS*, pages 211–226. Springer, 2005.

[94] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *International Journal on Software Technology and Tools Transfer*, 2006.

[95] K. Sen, A. Vardhan, G. Agha, , and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. In *26th International Conference on Software Engineering (ICSE'04)*, pages 418–427. IEEE, 2004.

[96] K. Sen, A. Vardhan, G. Agha, , and G. Roşu. On specifying and monitoring epistemic properties of distributed systems. In *2nd International Workshop on Dynamic Analysis (WODA'04), Satellite workshop of ICSE 2004*, pages 32–35. British Institution of Electrical Engineers (IEE), 2004.

[97] K. Sen, A. Vardhan, G. Agha, , and G. Roşu. Decentralized runtime analysis of multithreaded applications. In *NSF Next Generation Software Program Workshop (NSFNGS'06) (Satellite Workshop of IPDPS'06)*. IEEE Digital Library, 2006. (To Appear).

[98] K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *18th International Conference on Computer Aided Verification (CAV'06)*, LNCS. Springer, 2006. (To Appear).

[99] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 202–215. Springer, 2004.

[100] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 266–280. Springer, 2005.

[101] K. Sen, M. Viswanathan, and G. Agha. VESTA: A statistical model checker and analyzer for probabilistic systems. In *2nd International Conference on Quantitative Evaluation of Systems (QEST'05)*, pages 251–252. IEEE, 2005. Tool Paper.

[102] SGLIB. `http://xref-tech.com/sglib/main.html`.

[103] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. Technical Report UM-CS-2005-15, University of Massachusetts Department of Computer Science, 2005.

[104] G. L. Steele. Making asynchronous parallelism safe for the world. In *17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 218–231, 1990.

[105] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Fifth Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM Press, 1973.

[106] S. D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.

[107] M. Tatebayashi, N. Matsuzaki, and J. David B. Newman. Key distribution protocol for digital mobile communication systems. In *Proceedings on Advances in cryptology (CRYPTO '89)*, pages 324–334. Springer-Verlag, 1989.

[108] Valgrind. `http://valgrind.org/`.

[109] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON 1999*, pages 125–135, 1999.

[110] A. Valmari. Stubborn sets for reduced state space generation. In *10th Conference on Applications and Theory of Petri Nets*, pages 491–515, 1991.

[111] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for fifo automata. In *24th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'04)*, volume 3328 of *LNCS*, pages 494–505. Springer, 2004.

[112] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 274–289. Springer, 2004.

[113] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, pages 45–60. Springer, 2005.

[114] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering*. IEEE Computer Science Press, Sept. 2000.

[115] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[116] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *17th IEEE International Conference on Automated Software Engineering*, 2002.

[117] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[118] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Procs. of TACAS*, 2005.

Author's Biography

Koushik got his B.Tech in Computer Science and Engineering from the Indian Institute of Technology Kanpur, India in 1999. He subsequently worked as a software engineer and as a middleware architect in two companies before joining the University of Illinois at Urbana-Champaign in 2001. His paper on concolic testing won the ACM SIGSOFT Distinguished Paper Award at ESEC/FSE'05. He received the C.L. and Jane W-S. Liu Award in 2004 for exceptional research promise and the C. W. Gear Outstanding Graduate Award in 2005 from the UIUC Department of Computer Science.