The Dissertation Committee for Richard Brian Kilgore
certifies that this is the approved version of the following dissertation:

Testing Concurrent Software Systems

Committee:

Craig M Chase, Supervisor

Jacob A Abraham

Ernest A Emerson II

Vijay K Garg

Aleta M Ricciardi

Testing Concurrent Software Systems

by

Richard Brian Kilgore, B.S. E.E., M.S. C.E.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2006

Testing Concurrent Software Systems

Publication No. _____

Richard Brian Kilgore, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Craig M Chase

Two approaches to testing concurrent software are presented. In the first, a system is assumed to contain a deterministic computation when correct, and I describe two testing algorithms to optimally achieve coverage of a testing metric involving racing pairs of messages. In the second approach, the system model is improved to allow additional nondeterministic behavior when it is either commutative in nature or localized in its effect. I present two sets of algorithms. The first detects whether or not a system is deterministic (i.e., no race conditions affect the computation's outcome). The second algorithm identifies localized non-determinism, and can be used to determine whether or not a system converges to a deterministic end.

Contents

Chapter 1

Introduction

1.1 The problem

"Does my software do what I want it to do?" This problem has held computer scientists' attention for decades. Including the brute force method of running your system under simulated load over and over; functional testing, in which each function provided by the system is tested at least once; coverage testing, which entails executing every statement or branch choice at least once; various data-flow-based methods; and mathematical verification using a calculus based on the constructs of the program specification, conventional software validation techniques were designed for sequential systems. In testing a sequential system, by controlling system inputs from data sources and external systems, we can expect the program under test (PUT) to behave deterministically, producing the same outcome each time when run multiple times.

With the acceptance of parallel computing, the events of a computation are not totally ordered or deterministic. Variance in load on the computation devices in the system, and on the communication conduits between these devices can cause computation events to be ordered differently with respect to one another in different executions. When events are re-ordered, their dependencies can cause them to behave nondeterministically.

1.2 Testing and Verification

There are several different approaches to validating the correctness of software systems. The most straightforward approach is simply to run the system, attempt to make it do everything for which it was designed, and observe the results to see if they are as expected. This approach is generally referred to as *testing* [71]. When testing, we hope to be able to observe bad behavior to discover the existence of errors in the software wherever they exist. When testing succeeds in finding a path through the software that exhibits bad behavior, the next step is to determine the nature of the error so that it can be corrected. The process of doing so is usually referred to as *debugging* [71]. While debugging, one generally runs the software through the path in which the software behaves badly in slow motion: stopping and starting execution, and querying the *debugger* about the system's state at each stopping point. By narrowing the search during successive executions, it is generally possible to discover the nature of the error.

Another approach to validating a software system is to attempt to *prove* its correctness either by using a calculus to construct theorems about the system's behavior, or by systematically exploring the system's "potential behaviors" and verifying certain properties with predicate detection algorithms. In either approach, one generally uses a specification language to express properties that the system either must satisfy in all executions or should not satisfy in any execution.

The former verification approach is called *theorem proving* ([25, 62, 81, 45, 53]), and the latter is called *model checking* ([17, 88, 59, 18]). They are both *software verification* methods. Researchers have discovered that it generally helps to divide the properties of interest into two groups: safety and liveness properties. A safety property is used to express the possibility that the system takes a step in the wrong direction, or makes a bad decision or calculation. Typically, one uses the negation of such an expression as a safety property, and then attempts to verify that the system satisfies the safety property. A liveness property is used to verify that the system does not stall. Since the system can satisfy safety properties by simply doing nothing, we need a method to ensure that it makes progress.

Liveness properties are of the form: "the system eventually succeeds at completing task A" or "upon reaching state σ, the system will reach state σ'". Here, the term *eventually* represents the property that something will occur within a finite

(but unbounded) amount of time. For some applications, liveness properties can be specified using a language that bounds the amount of time, or the number of steps the system can take before making progress.

The prospect of *proving* that the system does what you want is an attractive one. Unfortunately, it is a difficult problem in general. Expressing the property "Does the system do what I want?" is sometimes impossible, when the phrase "what I want?" is either not completely understood or too complicated or unstructured to express with a specification language. After verifying that the system meets the specifications given, there is no way to actually verify that the specification is complete or correct with respect to what will be truly expected of the system in unanticipated scenarios.

Theorem proving typically requires manual proofs by skilled individuals with strong analytical abilities. Automatic theorem provers can help in the task, but still require interaction from the same skilled human theorem provers.

Regarding model checking, the straightforward approach of searching all possible execution paths through a software system can yield verification run times that are either immense or infinite. I discuss this problem, known as the *state explosion* problem, in chapter 2.

1.3 Concurrent Systems

The difficulty inherent in testing and debugging concurrent systems is that the system does not yield the same behavior during multiple executions. Race conditions cause different behavior to be realized during each execution, so one cannot expect to uncover errors by supplying a set of inputs only once, nor reproduce a previously detected error simply by supplying the same inputs.

An approach to allow debugging to proceed is to record the outcome of race conditions during the first execution (in which software testing identified the presence of an error), and control the outcome of these race conditions during subsequent executions to ensure reproducibility [37, 93, 76, 75]. If re-execution from the system's initial state is prohibitively costly, research has been done on the process of taking checkpoints, or snapshots of a system, from which the entire system's state can be restored and execution resumed from this point [5, 13, 50, 77, 91, 29].

Checkpoints and re-execution can also be employed to explore the effects of

race conditions in testing [96, 69, 38]. I use this approach to test a restricted class of distributed systems in the work described in Chapter 3.

Whether testing, debugging, or verifying concurrent software, modeling the system as a partial order of events can avoid some of the state space explosion that results from the interleaving of independent events. Lamport [57] first introduced the *happened-before* relation for expressing this partial order, and Fidge and Mattern independently discovered an efficient structure for holding the partial order [32, 63].

1.4 Overview

In the remaining chapters, I present methods and algorithms for validating concurrent software systems. In each chapter, I define a notion of correct behavior for the PUT that precludes certain types of nondeterministic behavior. In each case, correct behavior is expressed as a set of properties that allow load balancing on system resources, while at the same time constraining the nature of nondeterministic events to make testing or verifying the system a more realizable goal.

In chapter 3, my distributed system model is a set of communicating sequential processes [45], as is used to create concurrent programs in the MPI and PVM cluster programming environments. Processes explicitly address and pass messages to one another to communicate, and `recv` events block, causing synchronization between the sender and the receiver. For a given set of external input values, a process in a correct system is expected to send and receive the same messages in any execution, regardless of the order in which it receives racing messages. This constraint allows design using scatter-gather communication patterns, and unrestrained communication in which no races occur.

The testing algorithms defined in chapter 3 systematically re-order racing messages, searching for a feasible execution in which the PUT either creates externally visible outcomes that do not match expected behavior, or variant message-passing behavior.

Chapters 4 through 7 use a concurrent systems model that is event-based. Rather than having a process that explicitly executes a `recv` statement, an underlying system automatically invokes a function when its input messages become available.

With communicating sequential processes, unrelated events are artificially

ordered if they happen to occur on the same process. This is necessary, because if errors exist in the control flow logic of a process, it might misinterpret messages received in an unanticipated order. Detecting this condition requires analysis of the control-flow logic in the process's specification, which is typically expressed with a Turing-complete language.

In my event-based model, if it is possible for a message to be consumed by a different event (and hence interpreted differently), that fact is externally visible, without the need to analyze the Turing-complete language used in the function specifications. So unrelated events can be re-ordered without fear that the concurrent program will perform a different computation if, in another execution, racing messages are created or consumed in a different order. As a result, additional load balancing can occur between different threads of computation without incurring race conditions that must be judged as harmful by an automated validation system.

I hypothesize that there exists an interesting class of systems that, when implemented correctly, require the use only of these innocuous load-balancing race conditions (chapters 4 and 5). And the concurrency model I use enables an automated detection system to identify them as innocuous, and consequently declare a PUT to be deterministic. In chapters 6 and 7, I introduce a notion of *localized nondeterminism*, and describe detection algorithms for verifying that a PUT with such behavior *converges* to a deterministic end.

Chapter 2

Related Research

In this chapter, I discuss the state of related research using the three software validation techniques discussed in Chapter 1: software testing, debugging, and verification.

2.1 Testing

In its most basic form, software testing involves running the *program under test* (PUT) over and over, supplying different inputs, and observing to see if it produces expected results. The "expected results" can take the form of a formal specification, or it can be something that is understood to be verified by an oracle. If we could run the PUT in every possible way it can run (i.e., with every feasible set of inputs and input values) and verify that it produces correct results in each case, we could achieve certainty that it is correct. In general, however, the set of all possible executions of which the system is capable can be prohibitively large and often infinite.

Various heuristics-based methods have been researched that are designed to identify a set of execution runs that explore all the "important differences" between the possible executions. Each such method identifies a set of executions called a *test set*. I will group software testing approaches into two categories: *white box* testing, and *black box* testing.

White box testing

The term *white box* testing is used to describe an approach where the implementation text of the PUT is used to determine the test set. It generally takes the form of a

case-based analysis of the text. For example, the PUT might take *this* path or *that* path, and to test the two possibilities you run the program twice, trying each path once.

Popular forms of white box testing include methods that attempt to:

- execute every statement in the PUT at least once
- execute every branch of an `if-then-else` statement or loop
- execute paths through the PUT, attempting to approach the fault detection effectiveness of executing every possible path

It is almost never possible to execute every path through a PUT, or even some single structural elements of the PUT. The reason is that to test every possible path through a loop structure, it is necessary to run a separate test for every different number of iterations the loop might make in a feasible execution. When testing a larger portion as a unit, loops can multiply (whether nested or not), causing the number of test cases required to grow exponentially in the number of loops whose control predicates are sufficiently independent. A common approach taken to approximate path coverage over loops is to test the boundary conditions, such as 0, 1, and many iterations through the loop [78].

Another selection technique, described in [48], chapter 5, is to examine the data flow relationships between the statements in the PUT. Then, if statement s defines a value for some variable x used later by s', make sure to include a test run that traverses s and then s' where this relationship is realized. This concept can be extended to exploring chains of data flow relationships as well.

Another testing methodology that can be used as a complement to the coverage testing described above is called *mutation testing* [24]. Mutation implementations are created that vary slightly from the PUT, and the effectiveness of the test set generated using other means (such as coverage testing) is evaluated based on whether it can *kill* all the mutant implementations. In order to kill a mutant, a test case must cause the mutant implementation to produce a different outcome from that produced by the real one. If one or more mutants have not been killed, more tests are needed.

In [16], the author proposes a test set based on the finite state machine (FSM) implementation of a system that has been specified only informally. If the implementation contains n states, the tester hypothesizes that the correct implementation

of the desired system contains no more than m states, where $m > n$. Using an oracle to verify the input/output sequences of the implementation, the test set calculated ensures that the implementation is correct if the tester's estimate of the size of a correct implementation was accurate. In [60], the authors improve the size of the test set slightly, and extend its use to testing nondeterministic FSMs[1].

Black box testing

Using black box testing [9], which is sometimes called functional testing or specification-based testing, the test set is both designed for and verified against a specification, such as a finite state machine (FSM) or a *specification language*. The goals are often similar to those for white box testing: cover the events, logical branches, and paths in the specification for the PUT. In his book, Beizer discusses these approaches, as well as data flow-based testing, transaction-flow testing for exploring the effects of race conditions in high-level system testing, and a specialized approach for systems specified using a FSM that achieves link coverage in the FSM graph. His approaches are based on heuristics and knowledge gained through practice.

The approach for testing FSM-specified systems in [4, 92] is slightly more rigorous. These authors define the notion of a unique input/output sequence (UIO) for each state in the specification. If the PUT is in state σ, and the UIO for state σ is performed, the specification FSM will respond with a different set of output symbols than it would were it in any other state. The testing approach then becomes: for each state in the specification, supply a sequence of inputs required to reach that state, and then apply the UIO for that state to ensure that the implementation reached the state we expected, and that it responds the same as does the specification. The UIO is not guaranteed to be unique for each state in the implementation, so the approach is still heuristic in nature.

Another branch of FSM-specified system testing research is represented by [30]. In this work, they provide an automated algorithm for creating test cases designed to test a given use case of the system. The test case, which is a FSM to be covered using methods like those described in [9], is created from a *test purpose* that represents the use case, and from the specification FSM.

The research described in [12] uses an approach validates a specification

[1]The approach for covering the possible outcomes of nondeterministic state transition is through ad-hoc repetition.

composed of temporal logic formulas. In this sense, it is similar to model checking, but they use brute-force repetition, rather than state space reduction or reachability analysis, to determine whether properties like *always P*, *sometimes P*, and *eventually P* are satisfied or violated.

A different approach to black box testing is taken by some researchers, in which the possible values of inputs to a system are partitioned, and representatives chosen from each partition for inclusion in a test run [80, 102, 103]. Using this technique, parameters and environment conditions are identified for each individually tested functional unit that affect the function's behavior. They are categorized to identify the significant *choices* that should be selected in test runs. And dependence between inputs is analyzed to determine the constraints that choices for one input has on the feasible choices for others. Finally, a test suite is designed, ideally to cover all categories for all inputs.

2.2 Debugging

Once the presence of an error has been detected, debugging is the process of determining the nature of the error [2, 1, 3]. For sequential systems, one can simply run the PUT for a failed test case in an environment that allows extra control and monitoring. Typically, one sets one or more *breakpoints* either near the section of code that is suspected to contain the error, or before all candidate sections. The PUT then executes up to the point where it reaches the breakpoint that was set, and then stops, allowing one to examine the internal state of the system and step it forward in a controlled fashion, monitoring state changes in the process. Sometimes it is advantageous to narrow the search by setting multiple breakpoints designed to partition the execution in order to determine which partition contains the error. On a subsequent execution, breakpoints need only be set within the partition that was identified during the previous execution.

2.3 Verification

As discussed in section 1.2, software verification is the technique of using a proof system or model checking to mathematically verify that a software implementation meets its specification. The goal is to show with finality that the system is correct,

instead of merely indicating so as is done with software testing. Whereas software testing typically can only prove that errors exist, verification can be used to prove that they do not exist.

Using model checking, temporal logic is a useful means of system specification that lends itself to verification [26, 18, 85, 7, 61, 101]. By modeling a system with a state graph or Kripke structure, it is possible to verify or disprove safety and liveness properties specified in one of several forms of temporal logic (e.g., PLTL, CTL, CTL*, Mu-calculus). The approach entails a search of the paths through the model. During the search, investigation along a given prefix path *pre* can be terminated whenever one of the following two conditions is reached:

- all fullpaths with prefix *pre* satisfy the temporal logic property that is the subject of the search
- no fullpath with prefix *pre* can satisfy the temporal logic property in question

Researchers in this area have been attempting to overcome a state explosion that occurs when building the model for large or concurrent systems. One approach to the problem is to apply reduction techniques, such as reduction by symmetry (either general symmetry or for a given temporal logic formula) [19, 27, 49], and partial order reduction [57, 64, 86, 104]. Another approach is symbolic model checking [10, 66] in which the state space is represented more concisely using Binary Decision Diagrams (BDD). The result, if successful, is that the search space is reduced, and larger systems can be verified.

When validating concurrent systems, partial order reduction can help significantly in reducing the state space that results from many different interleavings of independent events. To explore the possible behaviors of the system, it can be sufficient to check at least one interleaving sequence of events from each class of logically equivalent executions (depending upon the temporal logic formula being verified). An equivalence class is called a *trace* [64].

Partial order reduction in model checking first appeared in [97, 98] and independently in [39, 42], and continued in [99, 100, 41, 83, 46, 84]. A summary publication on the subject compares the methods in these papers [40]. At each state in a search through the execution state space one identifies a *persistent set* of events. A persistent set T satisfies the property that for any path that does not

contain events in T, the events in the path are all independent with those in T. Considering only events from T at each step in the search is sufficient to guarantee that the reduced state space contains at least one representative interleaving sequence for each trace [40, 83].

While early results only addressed systems with finite traces, McMillan introduced the use of *cutoff events* to construct a finite trace cover for a system with infinite traces [65]. Their work uses net unfoldings of petri nets. A net unfolding represents distinct firings of the same transition as separate transitions in the unfolding. Its structure corresponds to the partial order between events in the system's traces, and a cutoff event is one that creates a marking on the original petri net that is reachable with a shorter trace through the unfolding.

Recent work on identifying more general forms of predicates that are efficiently detectable using partial order reduction appears in [51, 52]. In addition to continuing the search for classes of predicates that are efficiently detectable, Kashyap and Garg improve the effectiveness of reduction when applied to distributed systems by exploiting the properties of a predicate under analysis with respect to the system's processes. When constructing a persistent set of events that must be explored at each recursive step, they exploit the nature of the distributed system to efficiently eliminate *cutoff* events in every *cutoff process*. Events from a cutoff process are labeled *ineligible* with respect to the current trace.

A related area of research concerns predicate detection in a single trace of a concurrent system. Among other uses, this form of detection can be used to detect when a live system has taken a misstep, or to detect the existence of a global system state during debugging [20, 36, 35, 14, 95]. If the system is designed to behave deterministically, single-trace detection can be used to determine whether it truly is deterministic [47, 21, 23, 22, 72]. I employ this strategy in chapters 4 and 5.

For a single computation, Chandy and Lamport showed how to detect stable predicates [13] in polynomial time. Chase and Garg [14] proved that the problem is NP-complete for general predicates, but identified other classes of predicates, such as *linear predicates*, and an algorithm for detecting them in polynomial time. Mittal and Garg [68] defined a *slice* of a computation. For a given partial-order computation, and predicate Φ, the slice is the smallest computation containing all consistent cuts for which Φ is satisfied. The reduction to this subcomputation can result in an exponential improvement in time and space when performing predicate

detection over the slice rather than the complete computation. Mittal and Garg identify a class of predicates called regular predicates for which the slice contains exclusively consistent cuts in which the predicate is satisfied, and show how to compute the slice for regular predicates as well as other classes of predicates, such as linear, post-linear, and co-regular predicates. And for a predicate composed of conjunctive ($\wedge$) and disjunctive ($\vee$) operations on predicates in these classes, they show how to compute an approximate slice.

Another area that includes software verification research is petri nets [70, 90]. Properties that can be verified when the qualities of a system are specified using petri nets include:

reachability e.g., is a given marking reachable?

safeness can the number of tokens on a place exceed 1

boundedness can the number of tokens on a place exceed k or grow indefinitely?

liveness can a transition be fired? Can deadlock be reached?

persistence for two transitions, does firing one disable another?

synchronic distance dependency analysis – how many times can t fire between firings of t'

fairness t can only fire a bounded number of times before t' fires

repetitiveness exists a sequence in which every transition occurs infinitely often

controllability any M' is reachable from any M

Analysis techniques include simulation of execution (similar to model checking), various forms of structural analysis, and graph reduction.

The research I present in chapters 4 through 7 is closely related to partial-order reductions for model checking and the single computation predicate checking research described above. In fact, my approach to exploring the effects of localized nondeterminism in chapter 7 uses the partial-order methods to reduce the searched state space.

The goal in these chapters is more restricted than those in the related research. I use a model that is more amenable to discounting race conditions as sources of nondeterminism. Race conditions that are simply a re-ordering of independent events are easier to detect, and therefore load balancing is allowed without incurring an uncertainty in the outcome. Given this ability, I then postulate that there is an interesting class of systems that can be designed either without the need for true nondeterminism (chapters 4 and 5) or with the need only for *localized nondeterminism* (chapters 6 and 7).

When localized nondeterminism exists, I identify a process for determining the "states" at which disparate traces converge, such that their suffixes are equivalent sub-traces. The process exploits the partial order between events in the system to identify a *boundary* that delimits the nondeterministic behavior. The boundary is a cut across the partial order, such that local states below (i.e., after) the boundary are converged local states. As with partial-order reduction in model checking, I avoid the need to enumerate all the possible states the system might reach (due to alternate interleavings of concurrent events).

Chapter 3

Testing Distributed Programs with Re-execution

3.1 Introduction

Finding bugs in a program is always difficult. Much research has been devoted to testing and debugging of sequential programs [1, 3, 2, 29, 82]. Generally, an error in a program can start a chain reaction of unexpected events that is only noticeable externally much later. Detecting such anomalies usually involves running the program in a debugger, which allows one to view the internal state of the program, and step it forward in a controlled fashion. The process used to locate the bug usually involves running the program over and over, learning more and narrowing the search each time, until the mistake is finally found.

Since a typical execution of the program does not generally use all features of an application or provide all extremes or different types of input values, software testing is often used in an attempt to find more obscure bugs (e.g., divide by zero when the equation for the divisor rarely evaluates to 0). Using a test suite of inputs, the program is run a number of times, checking program trace data and outputs against expected values.

In a distributed computation, the problem is much worse. The inputs in most sequential programs are controllable by the programmer. It is possible to force a sequential program to run exactly the same each time. Concurrent programs introduce the difficulty of race conditions. Consider a receive event on a process in a

distributed computation at which more than one message is a candidate for delivery. A non-deterministic choice is made by the program when it chooses to receive one of these messages. This choice may affect data in future messages, or it may affect the control flow and consequently which messages are sent by this process in the future.

A bug observed during one execution run, may not have any effect on the computation in a subsequent run. This type of non-deterministic behavior makes software testing more difficult. Checking the results of the test is no longer simply a task of matching outputs with expected results, because the program's trace data varies from one execution to the next.

Most distributed debugging research has focused on either visualization [8, 43, 56, 89], or allowing the programmer to control execution or re-execution [31, 43, 93], or presenting dependence analysis information [15]. Methods for reproducing a distributed computation using trace and replay mechanisms are well known [37, 58, 74]. In [72], given information from the programmer that specifies which message races in a program are intentional, the authors show an approach for reporting the first unintentional non-determinism in a message-passing program.

In Sections 3.6 and 3.7, we attempt to automatically search the program execution space, looking for bugs of a type more familiar to a typical programmer. By re-executing the program and imposing a strictly different message ordering, we attempt to force bugs in a distributed program to cause anomalies.

The brute force method of exploring all possible executions of a distributed computation has been effectively argued to be intractable [11, 73]. Thus, we would like to find a method to explore a smaller set of possible executions that somehow captures important characteristics of all (or many) executions in the complete set.

Our current approach is applicable to distributed programs whose semantics dictate a *fixed message set* (See Section 3.4.1). We believe that certain types of distributed applications meet these requirements (e.g. loosely synchronous, divide-and-conquer-style programs). Additionally, we believe that the theory will be extendable in future work to more general types of distributed programs. We attempt to reverse pairs of messages that raced during the original test run. The algorithm presented in Section 3.6 produces an optimal single test run. In other words, it creates a single test run that reverses as many pairs of racing messages as possible. The algorithm requires $O(k^2)$ execution time when run on a process that passes k

messages in a single execution run. It can be run in parallel on all processes in the system.

In Section 3.7, we present an algorithm for calculating a minimal set of test runs to reverse all message pairs. The number of test runs required to achieve this 100% coverage can be determined after the first test run.

3.2 Model

In this section we present our model of a distributed computation, and provide a few definitions and notational conventions.

3.2.1 Processes and Distributed Systems

In this thesis, we model a process as a program that performs Turing-style computations. It can output part of its local state by sending a message to another process in the system, and it can receive a message from another process as input. A distributed system is a set of processes that communicate with one another by sending and receiving messages to and from one another as they perform computations. A process is represented with the symbol P, and we use subscripts when we need to refer to more than one process at a time.

3.2.2 Test Runs

An execution run on a single process is modeled as a sequence of events. Each event is either a receive event, at which the process delivers a message from some other process in the system; a send event, at which the process sends a message to another process; or an internal event, at which the process's local state changes. We use the symbol T to represent such a sequence, with subscripts if needed. Hereafter, we refer to a sequence T on a single process as a *test run*. A test run must have an initial state (i.e., any prefix is finite).

Given a test run T, if T_{pre} is a prefix of T, then $T_{pre} \leq T$ (or $T_{pre} < T$ if it is a strict prefix). More generally, we call any sequence contained in T a sub-computation of T.

We assume that the program that creates a test run is *piece-wise deterministic*. In other words, the order in which messages are delivered by receive events in

the process is its only element of non-determinism. If the program is run twice, and it delivers the same messages in the same sequence, then it will undergo the same state changes, send the same messages, and attempt to receive the same messages.

A distributed computation has an initial state and a final state. The initial state is composed of all processes' initial states, and the final state is reached when all processes have terminated. The algorithms presented in this chapter are directly applicable to real-world distributed computations that exhibit this behavior, but they can also be applied to sub-computations of a non-terminating distributed computation.

3.2.3 Relations

Given two events e_1 and e_2 from test run T, $e_1 \prec_T e_2$ indicates that e_1 was executed before e_2, and $e_1 \preccurlyeq_T e_2$ indicates that they might also be the same event. Since T is a sequential execution on a single process, $\prec_T$ imposes total ordering over the events in T. When T is clear from the context, $\prec_T$, pronounced "precedes", can be represented simply by the symbol $\prec$. As in Lamport [57], there exists a partial order, $\rightarrow$, between events in a distributed computation. For any two events, e and e', $e \ \rightarrow \ e'$ iff:

1. $e \prec_T e'$, where T is a test run on a given process, or
2. e is a send event, at which message m is sent; and e' is a receive event, at which m is delivered, or
3. $\langle \exists e'' :: e \ \rightarrow \ e'' \wedge e'' \ \rightarrow \ e' \rangle$

As discussed in Lamport's paper, this "happens before" relation is also transitive and anti-reflexive. When $e \rightarrow e'$, we also say that e' *causally follows* e. If $e \not\rightarrow e'$ and $e' \not\rightarrow e$, then the two events are *concurrent* and may occur in either order during a given test run.

3.2.4 Notational Conventions

When a message has been sent, we say it is *available*. When a message is actually passed to an application process by completion of a **msg_recv**( ) call, we say that it

is *delivered*. Our testing algorithms predetermine which message to deliver to each **msg_recv**( ) call by the application.

As mentioned earlier, processes in the system will be represented with the symbol P and a test run by T. A message passed during a distributed computation will be represented by m, and send and receive events by s and r, respectively. In all cases, if we need to refer to more than one at a time, we will use subscripts and/or symbols like m'. In a test run, for a given message m, $m.s_T$ represents the send event in T at which m is sent. $m.r_T$ represents the receive event in T at which m is delivered. When the test run to which we are referring is clear, these events will be abbreviated as $m.s$ and $m.r$.

Another necessary distinction is between the original, non-deterministic test run, and a controlled test run. In the *original test run*, each process's behavior is chosen non-deterministically, as processes are allowed to progress at their natural speeds and deliver available messages in any arbitrary order. If we refer generically to a test run on some process P by the symbol T, then the *original test run* will be denoted T_o. The second type of test run is pre-planned by a testing algorithm, and realized as a debugger/tester forces the underlying computation to deliver messages in the planned order.

3.3 Race Sets

Both contributions in this thesis rely on the ability to determine which messages received during a test run race with one another. In this section we group messages into sets: one for each receive event in T_o. If two messages appear together in any of these sets, they race.

Here we need to introduce the concept of a *race set*. $r.raceset$ is the race set for receive event r. It contains all messages that might have been available for delivery immediately preceding the occurrence of event r during a given test run, depending upon the relative speeds of the processes and communication channels. First, we define the set $r.avail$ of all messages that can exist before the occurrence of r.

Definition 3.3.1 *Given T on process P and a receive event r on T, define r.avail as follows:*

$$r.avail \triangleq \{m : (m \text{ is sent to } P) \wedge r \not\rightarrow m.s\}$$

The expression $(r \not\rightarrow m.s)$ can be evaluated efficiently using vector clocks [32, 63].

Lemma 3.3.2 *r.avail is increasing over the sequence of receive events in a test run:*

proof:

$$\begin{aligned}
& m \in r.avail \wedge r \prec r' \\
\Rightarrow\ & \{ \text{ Definitions of r.avail and } \rightarrow \ \} \\
& r \not\rightarrow m.s \wedge r \rightarrow r' \wedge (m \text{ sent to } P) \\
\Rightarrow\ & \{ \rightarrow \text{ transitivity } \} \\
& r' \not\rightarrow m.s \wedge (m \text{ sent to } P) \\
\equiv\ & \{ \text{ Definition of r.avail } \} \\
& m \in r'.avail \ \blacksquare
\end{aligned}$$

■

Next we define *r.recvd* as the set of messages delivered before event r in test run T.

Definition 3.3.3 *Given T on process P and a receive event r on T, define r.recvd as follows:*

$$r.recvd \triangleq \{m : m.r \prec_T r\}$$

As represented in the next lemma, any message must be available before it can be received.

Lemma 3.3.4 *r.recvd* $\subseteq$ *r.avail*
proof:

$$
\begin{aligned}
& m \in r.recvd \\
\equiv \quad & \{ \text{ Definition of r.recvd } \} \\
& m.r \prec r \\
\equiv \quad & \{ \ m.s \rightarrow m.r, \text{ definition of } \rightarrow \ \} \\
& m.s \rightarrow r \wedge (m \ \textit{sent to} \ P) \\
\Rightarrow \quad & \{ \ \rightarrow \text{anti-reflexive } \} \\
& r \not\rightarrow m.s \wedge (m \ \textit{sent to} \ P) \\
\equiv \quad & \{ \text{ Definition of r.avail } \} \\
& m \in r.avail \quad \blacksquare
\end{aligned}
$$

■

And finally, *r.raceset* is just *r.avail* minus the messages in *r.recvd* (i.e., any message that can possibly be ready for delivery that has not already been delivered).

Definition 3.3.5 *Given T on process P and a receive event r on T, define r.raceset as follows:*

$$r.raceset \triangleq r.avail - r.recvd$$

Example: Consider a test run in which process P receives 5 messages: a, b, c, d, and e, in that order (see Figure 3.1). Note that after each of the send events, new messages are introduced into the *r.avail*, and hence *r.raceset*. This is because, these messages are causally related to earlier intervals of P's execution, but after the send event, P has reached an interval with which these message's send events are concurrent.

We need to calculate *r.raceset* for each receive event in a test run. As demonstrated by the example of Figure 3.1, each message first becomes available after some send event on P. Thus, the messages received by P during a test run can be partitioned into *groups*: a new (possibly empty) group after each send event. To calculate *r.raceset* for each receive event r in a test run T, we need only keep a log of these groups, and of the receive event $m.r_T$ for each message m. The groups

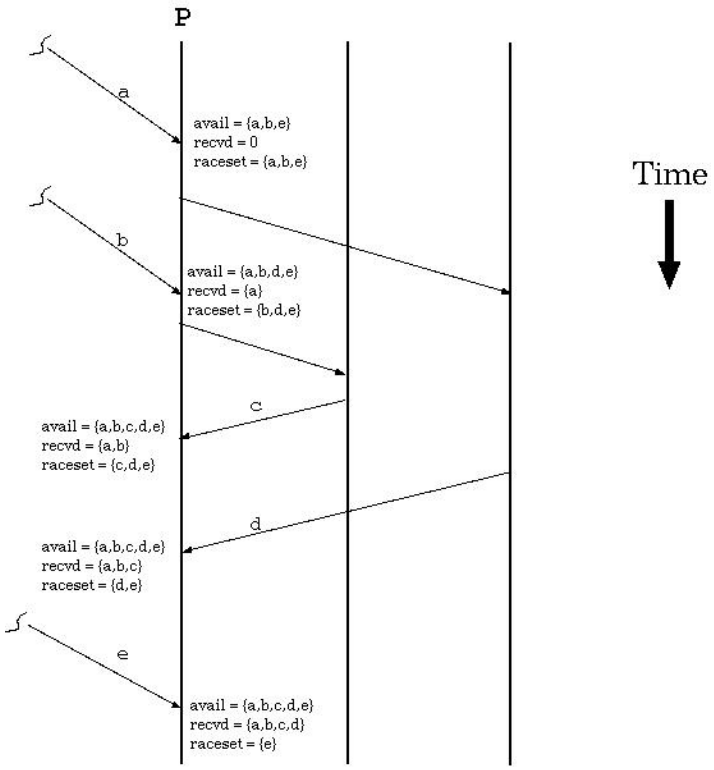

Figure 3.1: A Test Run and its Race Sets

are totally ordered and should be stored in order. After the test run (or during), we can analyze the receive events in order, and subtract messages from the groups when they are received. In other words, when calculating $r.raceset$ for receive event r, consider all groups for which the associated send event s satisfies $s \prec r$. Set $r.raceset$ to the sum of all such groups. Then, find the message in this set for which $m.r_T = r$, and remove it from its group.

Groups will be discussed in greater detail in Section 3.5.1. A detailed discussion of detecting race sets *on-line* can be found in [54, 55].

3.4 Pruning the Execution Space

Instead of involving a programmer with whether or not messages race, another approach is to design a software tester/debugger tool that can explore the effects of possible races automatically. The programmer might be more interested in the fact

that a task wait queue has exceeded some threshold, or that "index > array_size".

As discussed in Section 3.1, an approach to provide this capability is to prune the execution search space, and re-execute the program using pre-planned test executions. Our approach is to automatically explore alternate orderings on the delivery of pairs of messages that race, thus forcing test runs with different properties. Then, a user-specified predicate, like "$x > 10$" can be checked normally during these runs.

3.4.1 Fixed Message Set

Once the set of racing messages is determined for each receive event (see Section 3.3), we need to reorder them in some systematic fashion in an attempt to exploit software defects. The most immediate obstacle is the enormity of the search space. It is easy to imagine an imperfect program that never terminates when racing messages are delivered in a certain order.

If $|r.raceset| > 1$ for some receive event r on process P, then r can introduce non-determinism into the computation. If P received an alternate message at this receive event, it might act differently in the future. We restrict ourselves to distributed programs where the non-determinism introduced by a race does not change the affected process's set of future send and receive events. Regardless of the order in which messages were delivered to process P in its history, its underlying program will still evoke the same sequence of **msg_send**() and **msg_recv**() calls in the future. The data in these messages may be different, but send events will send to the same processes, and receive events will be the same requests. Thus, the set of messages sent by all processes in the distributed computation will also be the same, regardless of the order in which racing messages are delivered. We say that this kind of distributed program has a *fixed message set*. In such a system, messages can be reordered more easily in test runs.

A broad class of programs that typically exhibit this type of behavior are the "loosely synchronous" [33] parallel programs. Fox defined a loosely synchronous program as one that alternates between phases of global computation and communication. Typically, these programs send and receive the same messages during each communication phase, regardless of the results it obtained during the last communication phase. Non-determinism arises in such a program when they are executed

on asynchronous systems. Concurrent messages can race during a global communication phase.

This restriction affects our future discussion in a few ways. First, the set of all receive events in T_o is the same set as it is in any test run T. Further, the total order imposed by $\prec_{T_o}$ on the events in this set is identical to the total order imposed by $\prec_T$, for any test run T. The following lemmas are easily derived.

Lemma 3.4.1 *For a given receive event r in a program with a fixed message set, $r.avail_{T_o} = r.avail_T$.*

Lemma 3.4.2 *For a given receive event r in a program with a fixed message set, $|r.recvd_{T_o}| = |r.recvd_T|$, and $|r.raceset_{T_o}| = |r.raceset_T|$.*

3.4.2 Reordering Message Deliveries

To force different choices in a test run, we wish to reorder as many pairs of messages as possible. In other words, suppose messages m_1 and m_2 were delivered in T_o such that $m_1.r \prec_{T_o} m_2.r$. We want to construct a test run, T, in which the messages are delivered in the opposite order: $m_2.r \prec_T m_1.r$.

To construct T, we select a single message to deliver at each receive event r from its $r.avail$ set. From Lemma 3.4.1, $r.avail_T = r.avail_{T_o}$. The only care that must be taken, is that we cannot assign for delivery the same message by more than one receive event in T.

We can simply visit the receive events in the order in which they appear in T_o (and in T), assigning a message for delivery to each receive event in sequence. This way, we know the value of $r.recvd$ in T for each r evaluated, and we can avoid selecting the same message twice by simply selecting a message from only those in $r.avail - r.recvd_T$, (i.e., $r.raceset_T$).

Here, we introduce another type of sequence: a message sequence. The message sequence $M_{T_o} = \langle m_1 m_2 ... \rangle$ represents the scenario that message m_1 is delivered by the first receive event in T_o, m_2 is delivered by the second receive event, and so on. In this sequence, we use the notation $m_1 \prec_{T_o} m_2$ to express the condition that m_1 precedes m_2 in M_{T_o}. This predicate can also be represented using the more cumbersome expression $m_1.r_{T_o} \prec_{T_o} m_2.r_{T_o}$.

3.5 Groups, Funnels, and Waves

We number the receive events on a given process by their order in the execution: $\langle r_1, r_2, ... \rangle$. Messages are partitioned into **groups** by their first appearance in the $r.avail$ sets.

3.5.1 Groups

Using the numbering scheme described above for receive events, recall from Lemma 3.3.2 that

$$r_{x-1}.avail \subseteq r_x.avail$$

is always satisfied. In the following definition, a *group* is formed every time $r_{x-1}.avail \subset r_x.avail$.

Definition 3.5.1 *Wherever $r_{x-1}.avail \subset r_x.avail$, a* **group** *is formed, containing the set of newly available messages ($r_x.avail - r_{x-1}.avail$). The messages in $r_1.avail$ also form the first group (G_1) in the execution.* □

An example of groups will be given will be given later in this section (see Figure 3.3).

For group G, we name the first receive event at which the messages in this group are available $G.r_f$. The last receive event with the same $r.avail$ is denoted $G.r_l$. This event is either the last receive event in the execution, or it immediately precedes r_f of the next group.

Definition 3.5.2 $G.R \triangleq \{r : G.r_f \preccurlyeq r \preccurlyeq G.r_l\}$

As with receive events, we number groups by their order in the execution: $G_1, G_2, ..., G_g$, and we overload the $\prec$ operator over groups as follows:

$$G_i \prec G_j \triangleq G_i.r_l \prec G_j.r_f.$$

$G_i = G_j$ expresses the condition that these two symbols refer to the same group, and $G_i \preccurlyeq G_j$ is derived as expected.

As a result of the *fixed message set*, the set of receive events and $r.avail$ for each receive event remains constant between test runs. Thus, so does the set of

groups. Each receive event r in the subsequence $G.r_f \preccurlyeq r \preccurlyeq G.r_l$, has the same $r.avail$ set. We name this set $G.Avail$.

Definition 3.5.3 $G.Avail \triangleq \{m : \langle \exists G' \preccurlyeq G :: m \in G' \rangle\}$ □

3.5.2 Funnels

In a perfect test run, if $M_{T_o} = \langle m_1 m_2 ... m_k \rangle$, we would like to be able to deliver the messages in complete reverse order: $M_T = \langle m_k m_{k-1} ... m_1 \rangle$. Such a test run would successfully reverse every pair of messages. This is not always possible, as shown by the following example.

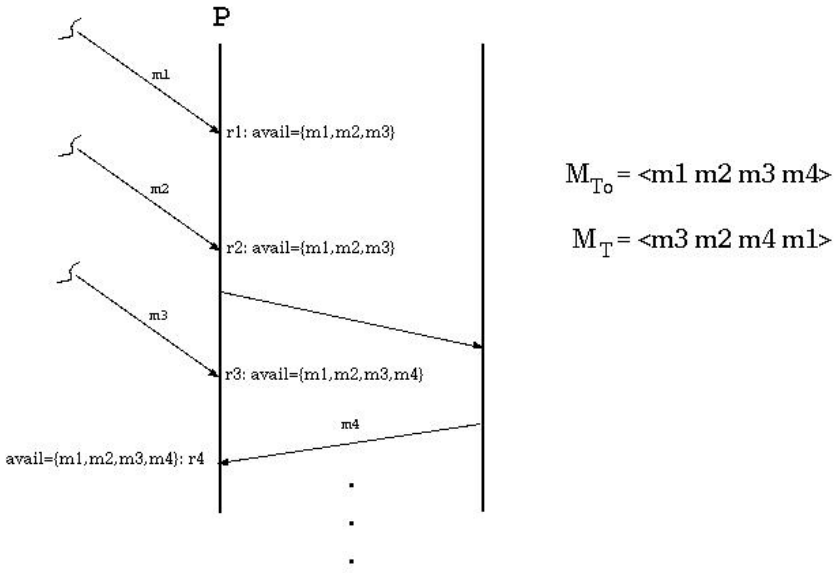

Figure 3.2: A $\mathbf{T_o}$ that Cannot be Reversed

Example: Consider the execution shown in Figure 3.2. We would like to deliver the messages in the order $\langle m_4 m_3 m_2 m_1 \rangle$. This sequence would reorder every pair of messages using only a single test run. However, since message m_4 is not included in $r_1.avail$ or $r_2.avail$, only messages from $\{m_1, m_2, m_3\}$ can be delivered by r_1 and r_2 in any test run. This reality is the consequence of a restrictive **funnel**, following the receive event r_2.

Definition 3.5.4 *For every group G_i, except the last one in the execution, there is a* **funnel**, *F_i, between receive events $G_i.r_l$ and $G_{i+1}.r_f$.*

The funnel has a throughput, given by the following equation:

$$F_i.tp = |G_i.Avail| - |\{r : r \preccurlyeq G_i.r_l\}|. \quad \square$$

We shall see in Section 3.7 that the key obstacle in reordering all messages in as few test runs as possible will be the need to deliver the messages in $G_i.Avail$ by receive events that follow $G_i.r_l$. We say that such a message is *passed through* the funnel.

Each $r \in \{r : r \preccurlyeq G_i.r_l\}$ must deliver a message from the set $G_i.Avail$ (since other messages are not available at these receive events). Thus, during a single test run, at most $|G_i.Avail| - |\{r : r \preccurlyeq G_i.r_l\}|$ (i.e., $F_i.tp$) messages from $G_i.Avail$ can be delivered after $G_i.r_l$. In other words, $F_i.tp$ messages *"pass through"* funnel F_i in a given test run.

In the discussion that follows, we include funnels in a test run's total order of events. Funnel F_i's position in the order is given by the following expression:

$$G_i.r_l \prec F_i \prec G_{i+1}.r_f.$$

3.5.3 Waves

We will show in this section that the messages received in a test run T_o, might be divided into *waves*. If two messages were delivered in different waves during the original test run, it is not possible to deliver them in the opposite order during a subsequent test run. Formally, if m_1 and m_2 were in different waves in T_o, and $m_1 \prec_{T_o} m_2$, then $\langle \nexists T :: m_2 \prec_T m_1 \rangle$.

Consider a loosely synchronous distributed computation, like the one described in Section 3.4.1. Suppose that at the end of every communication phase, the processes in the system perform a barrier synchronization. Process P cannot possibly deliver any messages sent in future communication phases during the current phase, because the other processes in the system will not advance to the point of sending these messages until P delivers all the messages in the current phase. Once P does so, its underlying program has no more receive events left in which a message from a future communication phase could be delivered.

Waves are formed by funnels that do not let *any* messages pass through. Each funnel F, whose $F.tp$ is 0 forms a wave boundary. In general, a program does

not have to be structured with explicit synchronization points for the messages it receives to be divided into waves. If a distributed computation simply does not contain very many message races, then most receive events will have a race set of size 1 (i.e., no race). This condition creates a funnel with zero throughput (i.e., a wave boundary).

Lemma 3.5.5 *Consider two messages m_1 and m_2 received by the same process during T_o. If there exists a funnel F_i, for which $F_i.tp = 0$ and $m_1 \prec_{T_o} F_i \prec_{T_o} m_2$, then m_1 and m_2* can not *be delivered in reverse order during any test run.*

proof: By contradiction.

Suppose that m_2 is delivered before m_1 in some test run T. Since $m_1 \prec_{T_o} F_i$, $m_1 \in G_i.Avail$. From our discussion of funnels, we know that m_1 can be delivered no later than receive event $G_i.r_l$ in any T. Thus, m_2 must be delivered by some $r', r' \prec_T G_i.r_l$. For this to be possible, we must have $m_2 \in G_i.Avail$. But this contradicts the fact that $F_i \prec_{T_o} m_2$, because no message from $G_i.Avail$ can pass through F_i. ■

In Lemma 3.5.5, funnel F_i forms a wave boundary. Messages available before F_i are separated from those that are not.

Lemma 3.5.6 *Consider two messages, m_1 and m_2, where $m_1 \prec_{T_o} m_2$, and*

$$\langle \nexists F_i : m_1 \prec_{T_o} F_i \prec_{T_o} m_2 : F_i.tp = 0 \rangle.$$

m_1 and m_2 can *be delivered in reverse order in some test run T.*

proof: Since there is no funnel F_i between $m_1.r_{T_o}$ and $m_2.r_{T_o}$ with $F_i.tp = 0$, m_1 can be passed through all funnels that do exist between them. At each such funnel F_j, $F_j.tp > 0$. Therefore, from the definition of $F_j.tp$,

$$|G_j.Avail| > |\{r : r \preccurlyeq G_j.r_l\}|.$$

Thus, $|\{r : r \preccurlyeq G_j.r_l\}|$ messages other than m_1 can be chosen from $G_j.Avail$ for delivery at the receive events in $\{r : r \preccurlyeq G_j.r_l\}$. Once m_1 has passed through the last funnel between $m_1.r_{T_o}$ and $m_2.r_{T_o}$, m_2 is in $r.avail$ for the very next receive event r in the test run. Thus, m_2 can be delivered by r and m_1 delivered by a later receive event. ■

In the lemma above, messages m_1 and m_2 are in the same *wave*, as are the receive events $m_1.r_{T_o}$ and $m_2.r_{T_o}$. We say that any pair of messages or receive events not separated by a funnel with zero throughput are *"in the same wave"*. Thus, from Lemma 3.5.6, any pair of messages in the same wave can be delivered in reverse order, and from Lemma 3.5.5, any two messages from different waves cannot.

3.5.4 Example Wave

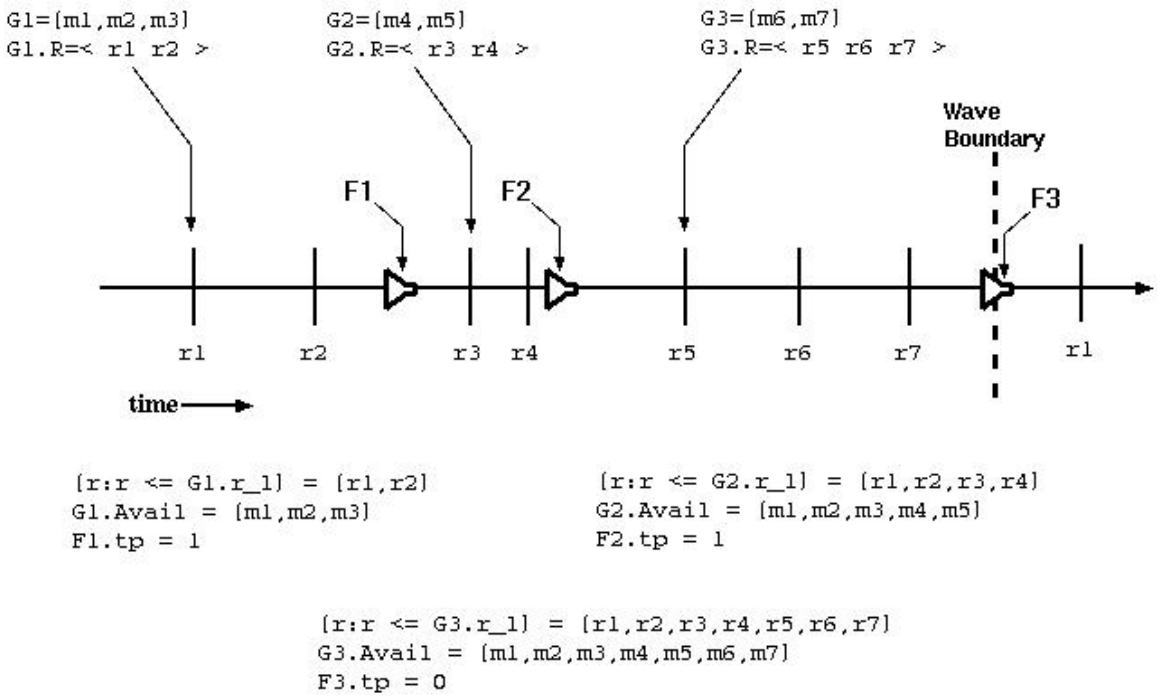

Figure 3.3: Groups and Funnels

3.5.5 Funnel and Wave Consequences

For the remainder of this chapter, our discussion is limited to the discussion of receive events and messages within a single wave. We define $G_i.Avail_W$ as the messages in $G_i.Avail$ that first become available in the current wave, W. In other words, if r' is the last receive event of the wave before W, $G_i.Avail_W = G_i.Avail - r'.avail$.

Since it is not always possible to deliver all messages in complete reverse order

(see Section 3.5.2), we will try to reverse as many pairs of messages as possible.

As we discussed earlier, at most $F_i.tp$ messages from $G_i.Avail_W$ can be delivered after $G_i.r_l$. The messages delivered by receive events in $\{r : r \preccurlyeq G_i.r_l\}$ cannot be delivered *after* any message in a future group, such as G_{i+1}. Lemma 3.5.7 follows trivially from this observation, and the fact that all messages in $G_i.Avail_W$ must be delivered after the messages in future groups to achieve 100% message reordering coverage over wave W.

Lemma 3.5.7 *100% message reordering coverage over a wave containing funnel F_i will require at least $\left\lceil \frac{|G_i.Avail_W|}{F_i.tp} \right\rceil$ test runs.*

The critical funnel defined below imposes a lower bound on the number of test runs we will need in order to achieve 100% coverage.

Definition 3.5.8 *The funnel for which the ratio $\left\lceil \frac{|G_i.Avail_W|}{F_i.tp} \right\rceil$ is maximal is called the* **critical funnel***, F_c.*

3.6 Single Test Run

In this section, we present an algorithm for reversing as many message pairs as possible in a single test run. On the process P that we are testing, first we allow T_o to run non-deterministically. During T_o's execution, we observe the program's message set and wave boundaries, and then calculate $r.avail$ for all receive events on P.

3.6.1 Last-First Reordering Algorithm

A very simple algorithm, shown in Figure 3.4, can be used to reverse the greatest number of pairs possible in a single test run, T. In the algorithm shown in this figure, r_o and r_t are used as iterators to scroll through the receive events in T_o and T. The set r_t.recvd contains all messages that have been assigned for delivery by some receive event r in T, $r \prec r_t$. Since Last-First( ) subtracts this set from the messages it considers for delivery at each step, it will never deliver the same message twice.

```
TestRun Last-First(TestRun T_o)
{
    TestRun T := T_o;
    RecvEvent r_o := FirstRecv(T_o);
    RecvEvent r_t := FirstRecv(T);
    SetofMsgs r_t.recvd := ∅;

    /* Note: Messages in sets sorted by
            total order (m.r, ≺_{T_o}). */
    /* r_o.avail is calculated on T_o from
            Definition 3.3.1. */
    /* r_t.recvd is r.recvd_T */

    while(r_o != NULL)
        r_t.raceset := r_o.avail − r_t.recvd;
        msg m := tail(r_t.raceset);
        deliver(r_t, m);
        r_t.recvd := r_t.recvd ∪ {m};
        r_o := NextRecv(T_o);
        r_t := NextRecv(T);
    endwhile

    return T;
}
```

Figure 3.4: Procedure Last-First( )

The set $r_o.avail$ is calculated as described in Section 3.3, and from Lemma 3.4.1 we know that $r_o.avail = r_t.avail$. This set is sorted by the order in which the messages were received in T_o. Thus, from the set of messages in $r_o.avail$ that have not already been assigned for delivery by an earlier receive event in T, m gets the one that was delivered last in T_o. In other words, for any pair of messages (m_1,m_2) in $r_t.raceset$ where $m_1 \prec_{T_o} m_2$, Last-First( ) will select m_2 before it selects m_1.

An efficient implementation of Last-First( ) would actually keep only the set of messages that become available for the first time at each receive event, and add them to some set S at each iteration of the while loop. Also, instead of keeping r_t.recvd, we can just subtract each message from S as it is received. If we use

a balanced tree to store the messages in S, the Last-First( ) algorithm runs in $O(k \cdot log\ k)$, for a sequence T_o with k receive events.

3.6.2 Last-First Analysis

To see that Last-First( ) is optimal, let's first examine the properties of the messages in the wave, and the effects of delivering one message instead of another at each receive event.

Definition 3.6.1 *For a given test run T, we define the set $m.lost_T$ for each message m in the wave:*

$$m.lost_T \triangleq \{m' : (m \prec_{T_o} m') \wedge (m \prec_T m')\}$$

Each message m' in $m.lost_T$ represents a pair (m, m') that is *not* reversed in T. The name *lost* refers to the lost opportunity to reverse these pairs. In a wave of size k, there are $\binom{k}{2}$ pairs. An optimal test run reverses the most pairs possible, so it contains the fewest *lost* pairs. For a test run T, the set of all lost pairs is given by:

$$T.lost \triangleq \{(m, m') : m' \in m.lost_T\}$$

Thus, an optimal test run contains a minimal $T.lost$ set.

We can now present an important lemma that is integral to proving our algorithm optimal.

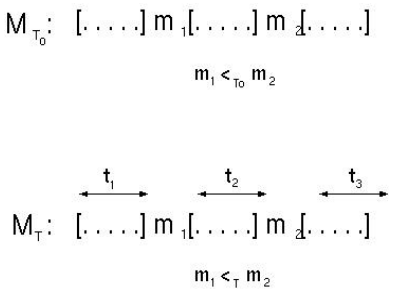

Figure 3.5: T - A test run that can be improved

Lemma 3.6.2 *Consider an original test run over a wave, T_o, and a test run T for the same wave. If T contains a pair of messages, (m_1, m_2), such that*

1. $m_2 \in m_1.r_T.avail$ *(i.e., m_1 is delivered by r in T, and $m_2 \in r.avail$)*
2. $m_2 \in m_1.lost_T$

then there exists a test run T', such that $|T'.lost| < |T.lost|$.

proof: We construct T'.
Construct T' as follows: duplicate T, and then swap m_1 and m_2 in $M_{T'}$. Since $m_2 \in m_1.r_T.avail$ (1 above), T' is a valid test run.

The message sequence M_T is shown in Figure 3.5. The messages in the sequences t_1, t_2, and t_3 are in the same positions in $M_{T'}$. To prove that $|T'.lost| < |T.lost|$, we compare the sets $m'.lost$ for every message m' in both test runs.

Observe that condition 2 in the lemma statement, implies that $m_1 \prec_{T_o} m_2$.

For every message m' in t_1 or t_3, pairs of the form (m_1,m') and (m_2,m') are ordered the same in M_T as they are in $M_{T'}$. Thus

$$\langle \forall m' \in t_1 \cup t_3 :: m'.lost_T = m'.lost_{T'} \rangle \tag{3.1}$$

Consider the set $m'.lost$ for a message m' in t_2. At first glance, it appears it is possible for this set to be larger in T' than it is in T, because while $m_1 \notin m'.lost_T$, it is possible that $m_1 \in m'.lost_{T'}$. However, when this is the case, we can deduce the following:

$$\begin{array}{ll} & m_1 \in m'.lost_{T'} \\ \Rightarrow & \{ \text{ Definition of m.lost } \} \\ & m' \prec_{T_o} m_1 \\ \Rightarrow & \{ \ (m_1 \prec_{T_o} m_2) \text{ from lemma condition 2 } \} \\ & m' \prec_{T_o} m_2 \\ \equiv & \{ \ (m' \prec_T m_2) \text{ and Definition of m.lost } \} \\ & m_2 \in m'.lost_T \end{array}$$

Thus,

$$\langle \forall m' \in t_2 :: |m'.lost_{T'}| \leq |m'.lost_T| \rangle \tag{3.2}$$

Now we must consider $m_1.lost$ and $m_2.lost$. $m_2.lost$ will possibly be smaller in T than in T', because for each message m' in t_2, it is possible that $m' \in m_2.lost_{T'}$, but we know $m' \notin m_2.lost_T$, because $m_2 \not\prec_T m'$. However, for every such m', we know the following:

$$\begin{array}{ll} & m' \in m_2.lost_{T'} \\ \equiv & \{ \ (m_2 \prec_{T'} m') \text{ and Definition of m.lost } \} \\ & m_2 \prec_{T_o} m' \\ \Rightarrow & \{ \ (m_1 \prec_{T_o} m_2) \text{ from lemma condition 2 } \} \\ & m_1 \prec_{T_o} m' \\ \equiv & \{ \ (m_1 \prec_T m') \text{ and Definition of m.lost } \} \\ & m' \in m_1.lost_T \end{array}$$

Also, $T.lost$ contains one additional pair, because $m_2 \in m_1.lost_T$, and $m_1 \notin m_2.lost_{T'}$. Thus,

$$|m_1.lost_{T'}| + |m_2.lost_{T'}| < |m_1.lost_T| + |m_2.lost_T| \tag{3.3}$$

From formulas 3.1, 3.2, and 3.3, $|T'.lost| < |T.lost|$. ■

Theorem 3.6.3 *The Last-First( ) reordering algorithm is optimal.*

proof: By contradiction.

We prove the stronger condition that Last-First( ) finds the only optimal test run over a wave. Consider the original test run T_o. For the purpose of contradiction, assume there exists an optimal test run T over this wave that differs from T_{lf}, the one created by Last-First( ). In Figure 3.5, assume that m_1 is the first message in M_T that differs from $M_{T_{lf}}$. Label the receive event that delivers m_1 here r. m_2 from Figure 3.5 represents the message that T_{lf} delivers at r. m_1 and m_2 are both in $r.avail$. Further, since the Last-First( ) algorithm delivers m_2 at r, we know that $m_1 \prec_{T_o} m_2$, which implies $m_2 \in m_1.lost_T$.

From Lemma 3.6.2, there exists a test run T' such that $|T'.lost| < |T.lost|$, so T is not optimal: a contradiction. ■

3.6.3 Performance Lower Bound

Following the pattern of logic in Lemma 3.5.6, it is always possible to reorder the message that was delivered by the first receive event in T_o, and deliver it in the very last receive event in T. Doing so reorders this first message with every other message in the wave, successfully reversing $k-1$ pairs. Since the Last-First( ) algorithm is optimal, $k-1$ represents a lower bound on the number of pairs it will reverse. Below we show this to be a tight bound.

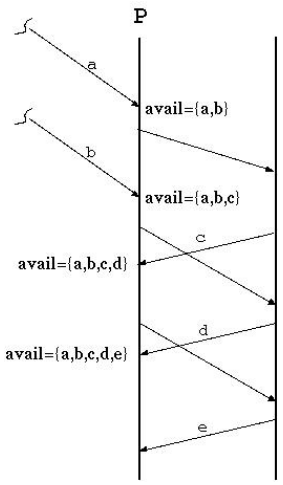

Figure 3.6: A Worst-Case Wave

Theorem 3.6.4 *$k-1$ is a tight lower bound on the number of message pairs the Last-First( ) reordering algorithm will be able to reverse in a wave of size k.*

proof: By example.

Consider the execution shown in Figure 3.6. For this rather extreme underlying program, our Last-First( ) reordering algorithm will only be able to reverse $k-1$ pairs. It produces the test run $T = bcd...a$, because it chooses any available message before a, but there is always only one other choice in $r.avail - r.recvd$. This test run reverses only $k-1$ pairs. ■

3.7 Creating a Test Suite

In this section, we present an algorithm that reverses all message pairs in a wave using the fewest number of test runs possible. Formally, our goal is to construct a *minimal* set of test runs, $T_1, T_2, ..., T_t$, such that for every pair of messages m_1 and m_2:

1. that are sent to process P,
2. that exist in the same wave, and
3. for which $m_1 \prec_{T_o} m_2$

the following proposition holds:

$$\langle \exists x : 1 \leq x \leq t : m_2 \prec_{T_x} m_1 \rangle.$$

In this section, we present a method for constructing such a minimal set. We call this set the test suite, and denote it T^*.

If no funnel exists, then all messages in the wave are included in $r_1.avail$. Only a single test run is required to reverse all pairs, using the Last-First( ) algorithm from Section 3.6.1. Assuming the wave contains at least one funnel, the following lower bound exists on the size of T^*.

Lemma 3.7.1 *100% message reordering coverage over a wave requires* **at least** $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$ *test runs.*
proof: From Definition 3.5.8 (the definition of F_c) and Lemma 3.5.7. ■

3.7.1 Multiple Test Run Algorithm

In this section, we present an algorithm for calculating a minimal T^*. Our method constructs one test run at a time: T_1, then T_2, and so forth. Each wave can be evaluated independently, so we discuss reordering the pairs within a given wave. The size of T^* is determined by the wave that requires the most test runs to reverse all its message pairs.

During the construction of T^*, we maintain a set of messages, $m.after$, for each message in the wave. While constructing test run T_x, for message m in wave

W, $m.after$ is the set:

$$\{m' : \langle \forall y : 1 \leq y < x : m \prec_{T_y} m' \rangle\}.$$

This set contains the messages that were delivered after m in every previous test run. If $m.after = \emptyset$, then m has been delivered after every other message in the wave during some test run. Thus, our algorithm continues to add test runs to T^* until $m.after = \emptyset$ for every message. The procedure **Create-Test-Run**( ) in Figure 3.7 is called to calculate each test run, T_x.

To construct T_x, **Create-Test-Run**( ) visits each group in order of the execution sequence. For each group, G_i, it first calls **Pass-Through**( ), which chooses the $F_i.tp$ messages that will be passed through funnel F_i. Then, it calls **Deliver**( ), which delivers the remaining available messages to the receive events in $G_i.R$.

```
SequenceOfMsgs Create-Test-Run( )
{
    SequenceOfMsgs TestRun := ⟨ ⟩;
    SetOfMsgs ChooseFrom := ∅;
    foreach (Group G_i)
    {
        ChooseFrom := ChooseFrom ⋃ G_i;
        SetOfMsgs PassThrough := Pass-Through(F_i, ChooseFrom);
        SequenceOfMsgs Seq := Deliver(ChooseFrom);
        TestRun := TestRun.Seq;  //Concatenation
        //Here, ChooseFrom = ∅
        ChooseFrom := PassThrough;
    }
    return TestRun;
}
```

Figure 3.7: Procedure Create-Test-Run( )

To determine which messages to pass through a funnel, **Pass-Through**( ) uses the pass through relation, $[PT]$.

Definition 3.7.2 *For every pair of messages (m, m') in the wave, where $m \in G$ and $m' \in G'$, define the relation $m[PT]m'$ as the disjunction of the following predicates:*

1. $(m.after \neq \emptyset) \wedge (m'.after = \emptyset)$

2. $((m.after = \emptyset) \equiv (m'.after = \emptyset)) \ \wedge$
   $(G \prec G')$

3. $((m.after = \emptyset) \equiv (m'.after = \emptyset)) \ \wedge$
   $(G = G') \ \wedge$
   $(m' \in m.after) \ \Box$

If $m[PT]m'$, then we want to pass m through a funnel more than m'. **Pass-Through**( ) maps $[PT]$ over the messages in the set $(G_i.Avail - G_i.r_f.recvd)$. Then, it removes $F_i.tp$ messages from the set, each one at the top of a chain formed by $[PT]$ over those messages that remain. These removed messages will be passed through F_i.

```
SetOfMsgs Pass-Through(Funnel F, SetOfMsgs ChooseFrom)
{
    SetOfMsgs PassThrough := ∅;
    while (|PassThrough| < F.tp)
    {
        Msg m := any message from ChooseFrom
        foreach (Msg m' ∈ ChooseFrom)
        {
            if (m'[PT]m)
            {
                m := m';
            }
        }
        PassThrough := PassThrough ⋃ {m};
        ChooseFrom := ChooseFrom − {m};
    }
    return PassThrough;
}
```

Figure 3.8: Procedure Pass-Through( )

To determine how to order the delivery of messages within some group G_i, **Deliver**( ) uses the simpler relation, $[D]$.

Definition 3.7.3 *For every pair of messages* (m, m') *in the wave:*

$$m[D]m' \triangleq m' \in m.after \quad \square$$

Consider two messages, m and m', where $m' \in m.after$. During some test run $T_x \in T^*$, we need to remove m' from $m.after$ by delivering the two messages such that $m' \prec_{T_x} m$. **Deliver**( ) visits the receive events in $G_i.R$ in order. When choosing between m and m' for delivery by a receive event, where $m[D]m'$, it delivers m' and leaves m in the set $ChooseFrom$ for delivery by a later receive event. This strategy is similar to that used by Last-First( ) in Section 3.6.

```
SequenceOfMsgs Deliver(SetOfMsgs ChooseFrom)
{
    SequenceOfMsgs Order := ⟨ ⟩;
    while (ChooseFrom ≠ ∅)
    {
        Msg m := any message from ChooseFrom
        foreach (Msg m'∈ChooseFrom)
        {
            if (m[D]m')
            {
                m := m';
            }
        }
        Order := Order.m;  //Concatenated
        ChooseFrom := ChooseFrom − {m};
    }
    return Order;
}
```

Figure 3.9: Procedure Deliver( )

3.7.2 Algorithm Analysis

In this section, we prove that our algorithm calculates a minimal T^*.

Definition 3.7.4 *For a group G_i, consider the set of funnels $\{F_j : G_i \preccurlyeq G_j\}$. Define $G_i.bn$ as the funnel from this set with the smallest throughput.*

The funnel $G_i.bn$ is the **bottleneck** for G_i. Later, we will see that this bottleneck defines an upper bound on the number of test runs required by our algorithm to reduce $m.after$ for every message in group G_i to $\emptyset$. For now, we note that whenever there are at least $G_i.bn.tp$ messages in G_i with $m.after \neq \emptyset$, $G_i.bn.tp$ messages from this set will be passed through all funnels in the wave.

Lemma 3.7.5 *While constructing test run T_x, consider group G_i.*

Let $S = \{m : m \in G_i.Avail_W \wedge m.after \neq \emptyset\}$
Let $R = \{m : m \in G_i \wedge m.after \neq \emptyset\}$

Either $G_i.bn.tp$ messages from S are passed through all future funnels, $\langle F_i, F_{i+1}, ..., F_{last}\rangle$, or **every** *message in R is passed through all future funnels.*
proof: By induction over the sequence of funnels.
Base: Funnel F_i. There are two cases:

> **Case 1:** There are at least $G_i.bn.tp$ messages from S still available. This means that at least $G_i.bn.tp - |R|$ messages with $m.after \neq \emptyset$ were passed through funnel F_{i-1} by **Pass-Through( )**. We will show that $G_i.bn.tp$ messages from S are passed through F_i. Suppose that this is not the case. Then some message m' from S was passed to our **Pass-Through( )** procedure and not included in the returned set. Also, since $F_i.tp \geq G_i.bn.tp$, at least one message m'', for which $m''.after = \emptyset$ **was** returned by **Pass-Through( )**. This is a contradiction, because **Pass-Through( )** will include m' in the return set before m'' in accordance with the $[PT]$ relation.
>
> **Case 2:** Fewer than $G_i.bn.tp$ messages from S are available. In this case, every message in R is passed trough F_i. The proof is similar to **Case 1**.

Step: F_j, for $i < j \leq last$.
By hypothesis, either at least $G_i.bn.tp$ messages from S were passed through F_{j-1}, or all messages in R were.

Case 1: At least $G_i.bn.tp$ massages from S were passed through F_{j-1}. In this case, we show that at least $G_i.bn.tp$ messages from S are passed through F_j. Assume this is not the case. Then some message $m' \in S$ is passed to **Pass-Through( )** and not returned. Also, some message $m'' \notin S$ **is** returned by **Pass-Through( )** for funnel F_j. Since $m'' \notin S$, it is either the case that $m''.after = \emptyset$, or $m'' \in G_k \wedge G_i \prec G_k$. In either case, the $[PT]$ relation causes **Pass-Through( )** to include m' in its return set before m'': a contradiction.

Case 2: All messages in R were passed through F_{j-1}. If $|R| > G_i.bn.tp$, then **Case 1** applies. Otherwise, all messages in R are passed through F_j. The proof is similar to **Case 1**.

■

The following lemma uses the properties of $[D]$ to show that once a message is passed through all funnels in the wave, its $m.after$ set is reduced to $\emptyset$.

Lemma 3.7.6 *If m is passed through every funnel in T_x, then $\langle \forall m' \in m.after :: m' \prec_{T_x} m \rangle$*

proof: By contradiction.
Suppose that m is passed through every funnel in T_x, $m' \in m.after$, and $m \prec_{T_x} m'$. Then m' was also passed through every funnel, and $m' \in m.r_{T_x}.raceset$. But this observation leads to the contradiction that Deliver( ) will deliver m' instead of m, because $m[D]m'$. ■

By combining the results from Lemmas 3.7.5 and 3.7.6, we can obtain an upper bound on the number of test runs our algorithm will require to reduce a given message's $m.after$ set to $\emptyset$.

Lemma 3.7.7 *Consider message m_x in group G_i. Within $\left\lceil \frac{|G_i.Avail_W|}{G_i.bn.tp} \right\rceil$ test runs, $m_x.after$ will be reduced to $\emptyset$.*

proof: From Lemma 3.7.5, we know that $G_i.bn.tp$ messages from $G_i.Avail_W$ with $m.after \neq \emptyset$ will be passed through all funnels during any test run in which m_x is not. Then, from Lemma 3.7.6, we know that $m.after$ is reduced to $\emptyset$ for each of these messages. Again from Lemma 3.7.5, once there are fewer than $G_i.bn.tp$ messages in $G_i.Avail_W$ with $m.after \neq \emptyset$, m_x will have to be passed through all funnels, and $m_x.after$ will consequently be reduced to $\emptyset$. In this worst case scenario,

m_x is the last message from $G_i.Avail_W$ to have its $m.after$ set reduced to $\emptyset$. This scenario requires $\left\lceil \frac{|G_i.Avail_W|}{G_i.bn.tp} \right\rceil$ test runs. ■

And finally, if we find the message(s) for which this upper bound is the largest, we know the worst-case largest size of T^*, as calculated by our algorithm. Upon comparing this upper bound with the lower bound from Lemma 3.7.1, we find that they are the same.

Theorem 3.7.8 *The reordering algorithm reverses all message pairs in* $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$ *test runs.*

proof: Lemma 3.7.7 gives an expression for the worst-case number of test runs our reordering algorithm will require to reverse any given message m with all others in the wave. This expression is maximized by selecting a message from the group corresponding to the funnel F_c. The resulting expression is $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$. From Lemma 3.7.1, this expression is also a lower bound on the number of test runs required to achieve 100% message reordering coverage. Thus, our reordering algorithm optimally requires exactly $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$ test runs to reorder all message pairs. ■

3.8 On-Line Message Race Analysis and Visualization

In this section we discuss another application for our method for calculating race sets. One approach to helping a programmer debug a distributed program is to help him visualize the interaction between the processes as the program runs [8, 43, 56, 89]. The popular representation is to display a time line for each process, where time runs from left to right. As the process runs, the line grows. Messages sent between processes can then be represented by directed lines from the sender's time line to that of the receiver. Hereafter, we refer to this representation of a distributed computation as its graph, or distributed computation graph.

We propose to combine the method described in Section 3.3, which calculates the race set of messages for a given receive event, with the channel predicate detection techniques in [34]. In doing so, we can provide on-line analysis of the race sets at each receive event. In a visualization tool similar to XPVM, a programmer could then simply select a receive event from the distributed computation graph, and be presented with a list of the messages in that receive event's race set.

To use this technique effectively in an on-line tool, we must be able to detect when a race set is *complete*. In other words, for some receive event r on P, how do we detect the condition that no future messages received by P will be added to $r.R$?

Definition 3.8.1 *Consider a receive event r on process P, and a future event e, $r \prec e$. The set $r.R$ is* **complete** *at e iff $\langle \forall m : e \prec m.r : m \notin r.R \rangle$.*

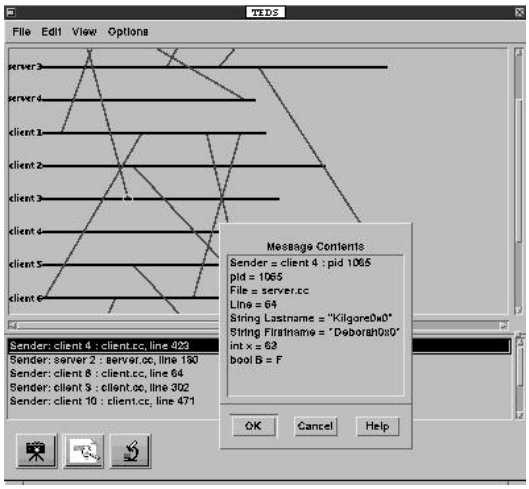

Figure 3.10: Race Detection and Visualization

The following lemma states that the race set $r.R$ is *complete* when the last event on every process in the computation causally follows r, and there are no messages in channels to P from send events that do not causally follow r. The "global state" G in the lemma is defined similarly to [34] and [36]. It is a set of: (a) local events, one from each process in the system, and (b) the state of all the channels.[1] $G[j]$ is the event from process P_j in G.

Lemma 3.8.2 *Consider a global state G, and a receive event $r \preccurlyeq G[i]$ on process P_i. $r.R$ is complete at G if:*

1. *$\forall m$ in a channel to P_i in G, $r \rightarrow m.s$*
2. *For every process $P_j \neq P_i$, $r \rightarrow G[j]$*

[1] In an alternative model used in [34, 36], G includes local states instead of events, each state immediately preceding an event on its process's computation.

proof: By contradiction.
Suppose that for global state G, conditions 1 and 2 from above are true, and that $\langle \exists m' :: G[i] \prec m'.r \wedge m' \in r.R \rangle$. Call the sender of message m' process P_j. Condition 1 can be stated formally as

$$\langle \forall m, j :: m.s \prec G[j] \wedge G[i] \prec m.r \Rightarrow r \rightarrow m.s \rangle.$$

Since $m' \in r.R$, we know that $r \not\rightarrow m'.s$ (See Definition 3.3.5). Thus, given our assumption that $G[i] \prec m'.r$, the contrapositive of condition 1 yields $G[j] \preccurlyeq m'.s$. But then, combining this result with condition 2, we have $r \rightarrow G[j] \preccurlyeq m'.s$, so $r \rightarrow m'.s$, and therefore $m' \notin r.R$: a contradiction. ■

As each new message is delivered, we can use the method from Section 3.3 to add it to race sets of previous receive events. Using the predicate detection algorithms from [34, 36] we can detect both conditions in Lemma 3.8.2. Thus, in a visualization application, the programmer could be alerted when race set $r.R$ is complete. Figure 3.10 shows a mock-up visualization application using the theory in this section. At the receive events on the computation graph, flashing vs. steady circles could indicate whether a message race set is complete.

3.9 Conclusion

We have presented an algorithm to reorder as many pairs as possible during a single test run, and another to reorder all pairs in as few test runs as possible.

The first area of research has yielded the Last-First( ) algorithm for reordering message deliveries in a test computation. We have proven that it is optimal, and given a tight lower bound of $(k-1)$ reversed pairs in a wave of size k. Questions remain that suggest opportunity for future research in this area. For example, the lower bound of $(k-1)$ is relatively low, but demonstrated by a particularly contrived underlying computation. We suspect that this bound is not indicative of performance for most real-world programs.

In Section 3.7, we presented an algorithm for calculating a minimal set of test runs to reverse all message pairs. The number of test runs required to achieve this 100% coverage, $\left\lceil \frac{|G_c.Avail_W|}{F_c.tp} \right\rceil$, can be determined after the first test run. Although

our model of a distributed system does not assume FIFO channels, this algorithm can be easily extended to accommodate such a system. The proofs in Section 3.7.2 are a little more complicated, but the performance of the extended algorithm is the same.

An interesting question is whether the techniques used in Sections 3.6 and 3.7 can be modified to allow testing programs that do not enforce a fixed message set. Perhaps a weaker requirement can be found.

Additionally, we intend to investigate alternate methods for reducing a distributed program's execution space to searchable subspaces. To evaluate any alternatives, we will also need a way to evaluate their abilities in exploiting software defects. Such an effort should most likely involve development of a prototype testing system for experimental analysis of their effectiveness. A taxonomy for software defects created and/or hidden by racing messages might also be useful to this end.

Chapter 4

Racing Writers

In this chapter, I explore an approach to limit the effect of *destructive race conditions*. First, by avoiding them altogether, and then by localizing consequences of race conditions. For the research covered in this chapter, if user-supplied inputs are fixed (including the initial state of the system and the values of inputs from the external environment), the system is expected to perform roughly the same computation during any feasible execution. In other words, outcomes of race conditions should not impact the system's outcome.

Section 4.1 describes the concurrent system model and its properties, section 4.2 defines a critical relationship between invocations that pertains to the system's and determinism, and section 4.3 introduces sufficient conditions for a deterministic system. Chapter 5 provides detection algorithms to determine whether a system meets the sufficient conditions for determinism.

In chapter 6, I add the notion of *localized nondeterminism*. A system with such nondeterminism, contains one or more race conditions that yield nondeterminism in the system. But the nondeterminism is localized, and the system *converges* to a deterministic end regardless of the paths chosen in the localized nondeterministic computation. Chapter 7 contains algorithms used to detect localized nondeterminism and convergence of the system.

4.1 Model

A system is composed of:

- a set X of system shared variables (shared between all threads of execution)
- a set F of named transition functions, called simply *functions*
- a changing set A of messages, each of which was produced by some $f_1 \in F$ and can be consumed by some $f_2 \in F$
- a set T of message types
- an initial state (X_0, M_0)

When it executes, a function from F consumes an ordered set of 1 or more messages from A, reads the value of 0 or more shared variables from X, assigns new values for 0 or more shared variables in X, and produces an ordered set of 0 or more output messages that are added to A. I use the notation $consume(i)$ to express the message input set, and $produce(i)$ for the message outputs. When these sets exist in an execution E, I write $consume(i, E)$ and $produce(i, E)$ to be explicit. If an invocation i reads the value of a shared variable x, I write $x \in use(i)$, and if it defines a value for x, then $x \in def(i)$.

Associated with each message is a type from T. A function specifies the messages it can consume by specifying a type for each input position. The consumed message's type must match the type for the input position into which it is consumed. Each input position must specify a different type, so that the assignment of messages available to input positions is unique.

Invocations are atomic. When two invocations cannot interfere with one another (by their shared variable uses and definitions), they can execute concurrently. But if they do interfere, the underlying system ensures their atomicity, either by employing synchronization or optimistic execution and rollback.[1]

An *execution* is a sequence of invocations, and a *feasible execution* is on in which every invocation consumes only messages available in A, and it produces output consistent with its underlying function and inputs.

Definition 4.1.1 - execution
An execution E *is a sequence of invocations:* $< i_1 i_2 ... >$.

I use the notation prefix(E, l) to represent a prefix of E of length l.

[1] See section 8.2 for a discussion about relaxing this requirement.

4.1.1 Deterministic Functions

Axiom 4.1.2 - deterministic functions
Given a function f and two invocations of that function i and i', if the portion of system state used as input by f in its computation is the same (i.e., system variables), and if the ordered set messages that f consumes are the same, then f performs the same computation and produces the same outputs in the two invocations.

In axiom 4.1.2, note that for the system state used by i and i' to be the same, it is not enough that it be the same when i and i' *begin* execution. For example, when i and i' read the value for some system variable x mid-way through their respective executions, the value each reads must be the same.

4.1.2 Identity

We need a way to uniquely identify a single function invocation that occurs in multiple feasible executions of the system. The same is true for the messages produced in different executions. When is a message in one execution considered to be the *same message* in another? When is a function invocation considered to be the same *invocation*?

I define their identities recursively, starting with the messages in M_0. In order to do so, I must introduce a restriction on M_0. I need to impose the restriction that no two messages in M_0 can have the same type. Thus, each can be uniquely identified by its type. The unique identities of messages and invocations are defined relative to one another.

Definition 4.1.3 message identity
Two messages that exist in separate system executions are the **same message** *iff:*

- *each is in M_0 and they have the same type, or*
- *each was produced in the same output position by the same invocation and they have the same type*

Definition 4.1.4 invocation identity
If the same function executes in two separate system executions, and if it consumes the same ordered set of messages, it is the **same invocation**.

4.1.3 Deterministic systems

If an invocation exists in all feasible executions (as defined recursively by definitions 4.1.3 and 4.1.4) and always behaves the same, I call it a *deterministic invocation*. If the system's feasible executions are composed entirely of deterministic invocations, then the system is deterministic.

Definition 4.1.5 - deterministic invocation
Invocation i is deterministic if it exists in every feasible execution, and it exhibits the same behavior *in every feasible execution, as follows:*

1. *it produces the same ordered set of output messages, containing the same data*
2. *it assigns to each system variable $x \in def(i)$ the same value*

Definition 4.1.6 - deterministic system
A system is deterministic iff every feasible execution contains only deterministic invocations.

Corollary 4.1.7 follows trivially from definition 4.1.6.

Corollary 4.1.7 *In a deterministic system, all feasible executions contain the same set of deterministic invocations.*

4.1.4 Event Ordering

When one invocation i produces a message, and another invocation i' consumes it, there is a causality relationship between i and i'. The consuming invocation cannot start execution until the producing invocation has executed and produced the message. Without the execution of i, i' cannot exist.

Definition 4.1.8 - $i \xrightarrow{m} i' \stackrel{def}{=} \langle \exists m \in M :: m \in produce(i) \cap consume(i') \rangle$

This relationship between i and i' is the building block for the $\xrightarrow{hb}$ relation, which is similar to causality relationships from [32], [57], and [63].

Definition 4.1.9 - $\xrightarrow{hb}$

For invocations i and i', I write $i \xrightarrow{hb} i'$, pronounced "i happens before i'", iff:

1. $i \xrightarrow{m} i'$, *or*
2. $\exists i''$ *s.t.* $i \xrightarrow{hb} i'' \wedge i'' \xrightarrow{hb} i'$

$\xrightarrow{hb}$ is asymmetric, transitive, and antireflexive. It forms a partial order over the events in an execution, E. Before an invocation i can occur in E, every invocation i' that *happens before* i must occur. In other words, the existence of i depends upon the existence and completion of i'. The $\xrightarrow{hb}$ relation can be represented by a directed, acyclic graph in which the nodes are invocations.

If we make a cut in the graph, partitioning it in two subgraphs, the cut is only consistent with the partial order if all of the arcs that span the cut (i.e., $\xrightarrow{m}$) traverse it in the same direction. If they do so, that is equivalent to the condition that in the partition on the source side of the cut, all dependencies are contained (definition 4.1.10).

Definition 4.1.10 - **consistent cut**

$C \subseteq I$ *is a consistent cut iff:* $\langle \forall i, i' \in I, i \xrightarrow{m} i' :: i' \in C \Rightarrow i \in C >$

4.1.5 Race Conditions

When two invocations are not related by $\xrightarrow{hb}$, they *race*.

Definition 4.1.11 - **races with**

i *races with* $i' \stackrel{def}{=} \neg(i \xrightarrow{hb} i') \wedge \neg(i' \xrightarrow{hb} i)$

When two invocations race there is potential for non-determinism, but only if they can affect each other's computations. Commonly called *interference*, in my concurrent system model, I call this condition a *shared variable conflict*, and write $svc(i, i')$.

Definition 4.1.12 - $svc(i, i')$

$$\begin{aligned} svc(i, i') \stackrel{def}{=} \langle \exists x \in X :: & \; x \in (def(i) \cup def(i')) \\ & \wedge \; x \in (use(i) \cup def(i)) \\ & \wedge \; x \in (use(i') \cup def(i')) \rangle \end{aligned}$$

When invocations i and i' race *and* there is a shared variable conflict, non-deterministic system behavior is a likely result. Thus, I say that i *races destructively* with i'.

Definition 4.1.13 - races destructively
i races destructively with i' $\stackrel{def}{=}$ $svc(i, i') \ \wedge$ *i races with i'*

The following lemma states that the $i \xrightarrow{hb} i'$ relation is preserved in all feasible executions that contain i'. It is derived from my definitions of message and invocation identity (4.1.3, 4.1.4).

Lemma 4.1.14 - preservation of $\xrightarrow{hb}$
Given two feasible executions E and E', if $i \xrightarrow{hb} i'$ in E, and invocation i' exists in E', then $i \xrightarrow{hb} i'$ in E'.
proof: $i \xrightarrow{hb} i'$ in E implies a path in E from i to i'. From definitions 4.1.3 and 4.1.4, the identity of i' is dependent on this path. This same path must exist in E', or i' would not be the same message. The existence of the path in E' implies that $i \xrightarrow{hb} i'$ in E'. ■

Lemma 4.1.14 proves that the *happens before* partial order is preserved between the invocations in other executions. If $i \xrightarrow{hb} i'$ in E, then i' must occur in any arbitrary execution E'

Another relation, $\xrightarrow{t}$, represents temporal ordering of events in a single feasible execution. It is a total order, consistent with $\xrightarrow{hb}$, and defined by the topological sort of invocations over $\xrightarrow{hb}$ that is imposed by the execution. Like $\xrightarrow{hb}$, $\xrightarrow{t}$ is also transitive, antireflexive, and asymmetric.

Definition 4.1.15 *For a given execution run, $i \xrightarrow{t} i'$ iff both i and i' are events in the execution, $i \neq i'$, and i executes before i'.*

The condition $i \xrightarrow{t} i'$ also represents the potential for i to affect the behavior of i' through manipulation of shared variables. If i executes before i', then i might define the value of some shared variable that i' uses. Note that $\xrightarrow{t}$ is a weaker relation than $\xrightarrow{hb}$: $i \xrightarrow{hb} i' \Rightarrow i \xrightarrow{t} i'$. And unlike $\xrightarrow{hb}$, it is not preserved between different executions. In a different execution, i' might occur before i, or even exist in the absence of i. In fact, if $\xrightarrow{t}$ is exactly the same between executions E and E', then E and E' are the same execution.

For the sakes of brevity and clarity, chains of $\xrightarrow{hb}$ or $\xrightarrow{t}$ relationships of the form:

$$i_1 \rightarrow i_2 \ \wedge \ i_2 \rightarrow i_3 \ \wedge \ ... \ \wedge \ i_{n-1} \rightarrow i_n$$

can be also expressed as

$$i_1 \rightarrow i_2 \rightarrow \ ... \ \rightarrow i_{n-1} \rightarrow i_n$$

4.1.6 Competing functions

Consider a feasible execution of the system that contains a snapshot in time in which A contains two intersecting sets of messages M_1 and M_2. M_1 is a complete set of input messages that could be delivered to some function $foo(\,)$, and M_2 is a complete set of input messages for some function $bar(\,)$.[2] If $m \in M_1 \cap M_2$, I say that $foo(\,)$ **competes** with $bar(\,)$ for message m in this execution. Only one of the two functions can invoke, and both are possible in alternate feasible executions. Function competition is a form of race condition.

4.2 Sole writers

When an invocation i uses a system variable x in a calculation, or if it reads the value of x to determine its control path, the value that x holds when it is read determines the behavior of i. If x does not hold the same value when read in two different executions, then i might perform a completely different computation or produce different messages or both. To ensure deterministic computation, the system must be designed and implemented to ensure that x does *not* contain different values when read by i in different feasible executions (except under controlled circumstances – Section 6).

To detect whether the system designer has accomplished this design goal, I wish to detect whether x will hold the same value across multiple executions, or whether it might contain different values, depending upon the outcome of one or more race conditions in the system.

If there is an invocation i_{dx} that defines x, and an invocation i that uses x, the relationship $i_{dx} \xrightarrow{hb} i$ ensures that x is initialized before i uses it. Since the

[2] $foo(\,)$ and $bar(\,)$ could also be the same function.

failure to initialize a variable is universally accepted as a software mistake in almost all cases, and since it is easy to detect this condition, I will assume from this point forward that at least one such i_{dx} exists.

In the relationship described above between i_{dx} and i, I call i_{dx} a *writer* of x, and i a *reader* of x. In equations, I express these conditions as $x \in def(i_{dx})$ and $x \in use(i)$, respectively. To determine if i will use the same value for x in all possible executions, we need to know how many different writers can potentially define the value for x that i uses. If only one writer is possible, and the same is true recursively for that writer and the variables it reads and so on, then we will see that i receives the same value for x in all executions.

Definition 4.2.1 - sole writer
For a given system execution E, i_{swx} is the sole writer of x for i iff:

$$x \in def(i_{swx}) \wedge i_{swx} \xrightarrow{hb} i \wedge \langle \forall i'_{swx} \neq i_{swx}, x \in def(i'_{swx}) :: i \xrightarrow{hb} i'_{swx} \vee i'_{swx} \xrightarrow{hb} i_{swx} \rangle \tag{4.1}$$

Lemma 4.2.2 shows that the **sole writer** of x for i is in fact, semantically, the sole writer of x for i. Lemma 4.2.3 extends the relationship between i_{swx} and i to all feasible executions. In executions other than E, i uses either the x value defined by i_{swx} or by some invocation that does not exist in E.

Lemma 4.2.2 - sole writer
If i_{swx} is the sole writer of x for i in E, then if i uses x in E, it uses the value defined by i_{swx}.

proof: The first two clauses of expression 4.1 ensure that i_{swx} defines x *before* i uses it. The universal quantification expression ensures that no other invocation can redefine x after i_{swx} and before i uses it in any execution. ■

Lemma 4.2.3 - sole writer in E'
If i_{swx} is the sole writer of x for i in E, then for any feasible execution E' in which i exists and uses x, i uses the value defined either by i_{swx} or some $i'_{swx} \notin E$.

proof: The first two clauses of expression 4.1 ensure that i_{swx} must exist in E' (preservation of $\xrightarrow{hb}$, lemma 4.1.14) and that it defines x *before* i uses it. Suppose

that some i'_{swx} defines x after i_{swx} does so and before i uses x. If i'_{swx} exists in E, then the preservation of $\xrightarrow{hb}$ (lemma 4.1.14) implies that

$$i \xrightarrow{hb} i'_{swx} \ \vee \ i'_{swx} \xrightarrow{hb} i_{swx}$$

not only in E (expression 4.1), but in E' as well. But either relationship conflicts with our supposition that allows i'_{swx} to define x for i in E'. Therefore, i'_{swx} must not exist in E. ■

Later, I will show that if there exists an i_{swx} that is the sole writer of x for i for every invocation i in the system and every variable x used by i, the system is deterministic. First, I need a few definitions regarding determinism for the functions, function invocations, and the system.

4.3 System determinism

From the concept of a sole writer (definition 4.2.1, lemmas 4.2.2 and 4.2.3), I can prove a system to be deterministic based on the properties of a single feasible execution. Theorem 4.3.1 proves that the existence of a sole writer i_{swx} for every invocation i and $x \in use(i)$ is sufficient to ensure a *deterministic system* (Definition 4.1.6).

Theorem 4.3.1 *If the following properties are true for a feasible execution E_0, then the system is deterministic.*

1. *for every invocation i and system variable $x \in use(i)$, there either exists a sole writer i_{swx}, or i deterministically uses the initial value for x (i.e., $\langle \forall i_{dx}, x \in def(i_{dx}) :: i \xrightarrow{hb} i_{dx} \rangle$)*
2. *no two functions ever compete to consume a message*

proof: *By induction*

For an arbitrary execution E of the same system, I show that $E \equiv E_0$.

Induction hypothesis - if the following conditions hold for prefix(E, l), then they hold for prefix$(E, l{+}1)$:

1. prefix(E, l) is a consistent cut from E_0
2. prefix(E, l) contains no invocations or messages that did not exist in E_0
3. every invocation $i \in$prefix(E, l) defines the same values to the same shared variables as in E_0
4. every invocation $i \in$prefix(E, l) produces the same messages with the same payloads as in E_0

base case: $l = 0$ - trivially satisifed
step: Invocation $l{+}1$ in E consumes messages from A. By the induction hypothesis, these messages all exist in E_0. And since there is no message competition in E_0, i must exist in E_0.

For each shared variable $x \in use(i)$, consider i_{dx}, the invocation that last defines x before i uses it. I show that $i_{dx} = i_{swx}$. Every other invocation i'_{dx} in prefix(E, l) that defines x also exists in E_0. And in E_0 it is known that $i'_{dx} \xrightarrow{hb} i_{swx} \vee i \xrightarrow{hb} i'_{dx}$. If the first clause holds, then i_{swx} defines a value after i'_{dx}. And if the second clause holds, then i'_{dx} is not in prefix(E, l).

Thus, all of i's inputs are the same in E as they are in E_0. So it will define the same values for shared variables and produce the same messages in E as it does in E_0. Therefore, the induction hypothesis holds for prefix$(E, l{+}1)$. ■

4.4 Sole Writer Properties

The following lemmas will be useful in the design of a testing tool in Chapter 5.

Lemma 4.4.1 *Suppose invocation i uses x and there is no sole writer i_{swx} that defines x for i. Further suppose that i does not use x's initial value (i.e., it is not the case that $i \xrightarrow{hb} i_{dx}, \forall i_{dx} : x \in def(i_{dx})$). Then there is a either race condition involving i and an invocation that defines x, or a race condition involving two invocations that define x.* **proof:** *By contradiction.*

Presume the lemma is false. Such an invocation i exists that has no sole writer i_{swx} and does not use x's initial value. From definition 4.2.1,

there is no writer i_{dx} for which the following expression holds:

$$i_{dx} \xrightarrow{hb} i \ \wedge \ \langle \forall i'_{dx}, x \in def(i'_{dx}) :: i \xrightarrow{hb} i'_{dx} \vee i'_{dx} \xrightarrow{hb} i_{dx} \rangle$$

So the opposite must hold for all $i_{dx} : x \in def(i_{dx})$:

$$\neg(i_{dx} \xrightarrow{hb} i) \ \vee \ \langle \exists i'_{dx}, x \in def(i'_{dx}) :: \neg(i \xrightarrow{hb} i'_{dx}) \wedge \neg(i'_{dx} \xrightarrow{hb} i_{dx}) \rangle$$

From the definitions for $\xrightarrow{hb}$ and *races with* (definitions 4.1.9 and 4.1.11), this expression can be transformed into:

$$\begin{aligned} & i \xrightarrow{hb} i_{dx} \vee i \ \textit{races with} \ i_{dx} \\ \vee \ \langle \exists i'_{dx}, x \in def(i'_{dx}) :: & (i_{dx} \xrightarrow{hb} i'_{dx} \vee i_{dx} \ \textit{races with} \ i'_{dx}) \\ & \wedge (i'_{dx} \xrightarrow{hb} i \vee i'_{dx} \ \textit{races with} \ i) \rangle \end{aligned}$$

Since we are presuming the lemma is false for our proof by contradiction, i does not race with i_{dx}, i_{dx} does not race with i'_{dx}, and i'_{dx} does not race with i. Substituting **false** for the three affected sub-expressions, we get:

$$i \xrightarrow{hb} i_{dx} \vee \langle \exists i'_{dx}, x \in def(i'_{dx}) :: i_{dx} \xrightarrow{hb} i'_{dx} \wedge i'_{dx} \xrightarrow{hb} i \rangle \tag{4.2}$$

Expression 4.2 holds for all $i_{dx}, x \in def(i_{dx})$. Again from our presumption that the lemma is false, $i \xrightarrow{hb} i_{dx}$ does not hold for all i_{dx}, because that would mean that i deterministically uses the initial value for x. Thus, there exists at least one i_{dx} for which:

$$\langle \exists i'_{dx}, x \in def(i'_{dx}) :: i_{dx} \xrightarrow{hb} i'_{dx} \xrightarrow{hb} i \rangle$$

Consider one such i'_{dx}. Since i'_{dx} also defines x, it too must satisfy expression 4.2 (re-written with $i_{dx} := i'_{dx}$, $i'_{dx} := i''_{dx}$):

$$i \xrightarrow{hb} i'_{dx} \vee \langle \exists i''_{dx}, x \in def(i''_{dx}) :: i'_{dx} \xrightarrow{hb} i''_{dx} \wedge i''_{dx} \xrightarrow{hb} i \rangle$$

And since $i'_{dx} \xrightarrow{hb} i$, we are left with:

$$\langle \exists i''_{dx}, x \in def(i''_{dx}) :: i'_{dx} \xrightarrow{hb} i''_{dx} \wedge i''_{dx} \xrightarrow{hb} i \rangle$$

thus identifying another writer i''_{dx}, and the cycle repeats. Note that the invocations identified so far satisfy $i_{dx} \xrightarrow{hb} i'_{dx} \xrightarrow{hb} i''_{dx} \xrightarrow{hb} i$. Since $\xrightarrow{hb}$ does not contain cycles, if we continue the analysis, we will identify an infinite $\xrightarrow{hb}$ chain from i_{dx} to i. Such an infinite chain cannot exist. Thus, a contradiction is reached. ■

Lemma 4.4.1 states that the lack of a sole writer implies existence of a destructive race condition (definition 4.1.13). The following two lemmas complete the picture by showing that existence of a destructive race condition implies the lack of a sole writer for some $x \in X$ and i_{ux} that uses it. These two conditions are equivalent.

Lemma 4.4.2 *Consider an execution E with two invocations i_{dx1} and i_{dx2}: both define x and they race. A third invocation, i_{ux}, uses the value of x defined by i_{dx1}. There is no sole writer of x for i_{ux}.*

proof: (by contradiction)

Suppose there *is* a sole writer. There are two possibilities: i_{dx1} is the sole writer, or it is not. If i_{dx1} is the sole writer, then:

$$i_{dx1} \xrightarrow{hb} i_{ux} \ \wedge \ \langle \forall i_{dx} \neq i_{dx1}, x \in def(i_{dx}) :: i_{ux} \xrightarrow{hb} i_{dx} \ \vee \ i_{dx} \xrightarrow{hb} i_{dx1} \rangle$$

Since $x \in def(i_{dx2})$, the universally quantified expression must hold for i_{dx2}:

$$i_{ux} \xrightarrow{hb} i_{dx2} \ \vee \ i_{dx2} \xrightarrow{hb} i_{dx1}$$

And since i_{dx2} races with i_{dx1}, $i_{ux} \xrightarrow{hb} i_{dx2}$ must hold. But then we have:

$$i_{dx1} \xrightarrow{hb} i_{ux} \xrightarrow{hb} i_{dx2}$$

a contradiction, since i_{dx1} and i_{dx2} race.

If the sole writer is another invocation, $i_{swx} \neq i_{dx1}$, then:

$$i_{swx} \xrightarrow{hb} i_{ux} \ \wedge \ \langle \forall i_{dx} \neq i_{swx}, x \in def(i_{dx}) :: i_{ux} \xrightarrow{hb} i_{dx} \ \vee \ i_{dx} \xrightarrow{hb} i_{swx} \rangle$$

This time, we apply the universally quantified expression to i_{dx1}:

$$i_{ux} \xrightarrow{hb} i_{dx1} \ \vee \ i_{dx1} \xrightarrow{hb} i_{swx}$$

The first disjunctive sub-expression here cannot hold, because i_{ux} uses the value of x defined by i_{dx1}, so clearly $i_{dx1} \xrightarrow{t} i_{ux}$. And the second sub-expression cannot hold, because it yields the relationship $i_{dx1} \xrightarrow{hb} i_{swx} \xrightarrow{hb} i_{ux}$, which means that i_{swx} redefines x after i_{dx1} and before i_{ux} uses it. Again, a contradiction. ■

Lemma 4.4.3 *Suppose i_{dx} and i_{ux} race, i_{dx} defines x, and i_{ux} uses x. There is no sole writer of x for i_{ux}.*
proof: From definition 4.2.1, a sole writer i_{swx} of x for i_{ux} must satisfy the following expression:

$$i_{swx} \xrightarrow{hb} i_{ux} \ \wedge \ \langle \forall i_{dx} \neq i_{swx}, x \in def(i_{dx}) :: i_{ux} \xrightarrow{hb} i_{dx} \vee i_{dx} \xrightarrow{hb} i_{swx} \rangle$$

i_{dx} cannot be a sole writer, because i_{dx} and i_{ux} race, so the first sub-expression ($i_{swx} \xrightarrow{hb} i_{ux}$) does not hold for $i_{swx} := i_{dx}$. Suppose there is an invocation i'_{dx} that is the sole writer. Substituting i'_{dx} for i_{swx} above, the resulting expression implies that:

$$i'_{dx} \xrightarrow{hb} i_{ux} \ \wedge \ (i_{ux} \xrightarrow{hb} i_{dx} \ \vee \ i_{dx} \xrightarrow{hb} i'_{dx})$$

Since i_{dx} and i_{ux} race, $i_{ux} \xrightarrow{hb} i_{dx}$ is false, and we are left with:

$$i'_{dx} \xrightarrow{hb} i_{ux} \ \wedge \ i_{dx} \xrightarrow{hb} i'_{dx}$$

which can be rewritten as:

$$i_{dx} \xrightarrow{hb} i'_{dx} \xrightarrow{hb} i_{ux}$$

but this expression contradicts the lemma's premise that i_{dx} and i_{ux} race. ■

In Chapter 5 I use the knowledge provided by lemmas 4.4.1, 4.4.2, and 4.4.3 to design a testing tool that detects destructive race conditions, rather than the absence of sole writers.

Chapter 5

Detection

In this chapter, I describe algorithms for detecting whether a system is deterministic. As discussed in sections 4.3 and 4.4, sufficient properties are the absence of destructive race conditions, and the absence of message competition.

5.1 Creating an execution graph

To develop race detection algorithms, I need to construct an *execution graph* (also known as a *causality graph*) for an execution E under analysis. To do so, the system is instrumented so that a monitor can observe interactions between invocations during runtime. The monitor assigns unique IDs to messages and invocations as it observes them.

When a message is produced, the monitor associates that message with its producing invocation (e.g., by storing it in a data structure indexed by the message's ID), and when the message is consumed the monitor associates it with the consuming invocation. These associations hold the structure of the execution graph. The invocations are stored in a structure Inv by their IDs, and the messages are stored in a structure $Msgs$, by *their* IDs. The associations between the entities must support querying for a message's producer and consumer invocations, and also querying for an invocation's consumed and produced messages (i.e., there are references stored in both directions).

5.1.1 Complexity

The variables in the complexity analysis that follows hold the following definitions:

m number of messages in E
i number of invocations in E
c number of messages consumed by an invocation (avg.)
p number of messages produced by an invocation (avg.)

The sizes of Inv and $Msgs$ are i and m, respectively, and there are $O(m)$ associations between invocations and messages. Using any efficient data structure, adding the invocations to Inv takes $O(i \cdot log(i))$ work and adding messages to $Msgs$ takes $O(m \cdot log(m))$ work. Retrieving an invocation or message is $O(log(i))$ and $O(log(m))$, respectively. Since $m > i$, overall size of the data is $O(m)$, and the total work of creating the execution graph is $O(m \cdot log(m))$.

5.2 Walking the graph

When traversing or searching through the execution graph, I will often want to *visit* an invocation node i only after all of i's immediate predecessors have already been visited (i.e., all messages in $consume(i)$ are produced by already visited invocations). I call this ordered traversal process *walking the graph*. To achieve it, I need to mark each message as *produced* once its producing invocation is visited, and I need to maintain throughout the traversal a list of *enabled* invocations. At any point in the traversal, the *enabled* list holds each unvisited invocation i for which $m \in consume(i)$ implies that m is *produced*.

When an invocation i is visited during a walk, each message m_p that i produces is marked *produced*. We then retrieve the invocation i' that consumes m_p and all the other messages consumed by i' (from the execution graph). If the state for every message in this consumed set indicates that the message is *produced*, then add i' to the *enabled* list.

To initialize, start with an empty *enabled* list and perform this same procedure for all messages in M_0.

5.2.1 Complexity

The variables in the complexity analysis that follows hold the following definitions:

m number of messages in E
i number of invocations in E
c number of messages consumed by an invocation (avg.)
p number of messages produced by an invocation (avg.)
e maximum size of *enabled* (could be $O(i)$, but is potentially less)

When invocation i is visited, retrieving and updating the status for each message m_p that i produces requires $O(log(m))$ work. For each m_p, retrieving the invocation i' that consumes m_p and retrieving the other messages consumed by i' takes $O(log(i))$ and $O(c \cdot log(m))$, respectively. Thus, performing these first few steps for every message produced in E requires $O(m(log(m) + log(i) + c \cdot log(m))) = O(mc \cdot log(m))$ time for a complete walk.

When the messages consumed by i' are all *produced*, we must perform $O(log(e))$ work to insert i' into the *enabled* list. This work is only performed once for each invocation (when the last message it consumes becomes *produced*), so the total work from this step is $O(i \cdot log(e))$. Adding this expression to the previous one, we obtain the total work required to walk the graph: $O(mc \cdot log(m) + i \cdot log(e))$. Since $e < i < m$, this expression reduces to just the first term: $O(mc \cdot log(m))$.

5.3 Race Detection

To find destructive races, I will need to identify which invocations in the execution race with one another (i.e., $\neg(i \xrightarrow{hb} i') \wedge \neg(i' \xrightarrow{hb} i)$), and among those, which ones interfere with each other by reading and writing the same shared variables. To answer the second part, I can either observe each invocation as it executes, recording the set of shared variables for which it consumes and produces values (i.e., $\{x : x \in use() \cup def()\}$), or I can perform static analysis on the program text for every function.

The former approach has the advantage of being more exact. Suppose static analysis predicts that some invocation i of a function f produces a value for the shared variable x, and i races with another invocation i' that also produces a value for x. But then suppose that i does not actually produce a value for x, because the

control flow of the function skips the instruction in f that does so. In this case, using static analysis, a race detection algorithm will incorrectly identify the race between i and i' as destructive.

If instead the algorithm dynamically determines $use()$ and $def()$ for i, it will correctly recognize the race as innocuous. But the disadvantage is that the detection algorithm will be more expensive. As a result, I choose instead to observe the widely accepted design principle that it is bad design for a function to be correct only by virtue of the fact that certain paths through its control flow are disabled when called with a given set of inputs. Accordingly, the algorithm I present here uses static analysis to determine the shared variables that each invocation produces and consumes.

5.3.1 Data structures

The first information I need is a mapping from each invocation in E to the function of which it is an invocation. I can add this relationship to the Inv data structure from section 5.1. Doing so will add only constant time additional computation for each invocation while computing Inv.

In addition, I need three new data structures.

Causality

For each invocation i in E, this structure holds i's causality history (i.e., $\{i' : i' \xrightarrow{hb} i\}$). It can be created by walking the execution graph (section 5.2) and calculating the history of each invocation i_v when it is visited by summing the histories of all its predecessors (i.e., every invocation that produces a message that i_v consumes).

FunctionRaces

Holds pair-wise races between functions. For any two invocations i_1 and i_2 that race in E, the pair of associated functions f_1 and f_2 is added to this data structure. *FunctionRaces* has size $O(f^2)$.

To populate *FunctionRaces*, query *Causality* for every pair of invocations. For each pair, if neither invocation is in the other's causality history, store the pair of associated functions.

If the goal of detection is to determine the yes/no answer to "Is there any destructive race?", *FunctionRaces* need only hold these

pairs of functions. If we also wish to identify *every* destructive race in E, each entry must hold a list of the associated racing invocation pairs.

UseDef

For every variable $x \in X$ and function $f \in F$, this data structures holds the value use, def, or use_def, to indicate whether f uses and/or defines x (i.e., whether it produces or consumes values for x). If f does not use or define x at all, then *UseDef* contains no entry for this combination. Create *UseDef* by performing static analysis on each function in F.

5.3.2 Detection Algorithm

Using the data structures above, we can find destructive races by iterating through the pairs of racing functions in *FunctionRaces* in an outer loop, and the variables in X in an inner loop. For each pair of racing functions and each shared variable x, check to see if one of the functions defines x while the other either uses or defines x. If so, a destructive race exists. To enumerate all existing races, use the lists of racing invocation pairs in *FunctionRaces*.

5.3.3 Race Detection Complexity

The variables in the complexity analysis that follows hold the following definitions:

m number of messages in E
i number of invocations in E
x number of shared variables in the system
f number of functions
c number of messages consumed by an invocation (avg.)
p number of messages produced by an invocation (avg.)
s size of a function's text

Causality

Constructing *Causality* requires a walk of the execution graph (shown to be $O(mc \cdot log(m))$ in section 5.2.1), and the work of building up the sets of causality histories. Combining histories for a single invocation i_v requires that we take the union of the

histories for i_v's c predecessors. Each history can have size $O(i)$, so we are iterating through $O(ci)$ invocations in the predecessors' histories, and adding them one by one to the history for i_v. The total work is $O(ci \cdot log(i))$. Thus, to do this work for all i invocations in E requires $O(ci^2 log(i))$, and the total cost of constructing *Causality* is $O(mc \cdot log(m) + ci^2 log(i))$.

FunctionRaces

In constructing *FunctionRaces*, each query to *Causality* to determine if $i \xrightarrow{hb} i'$ requires $O(log(i))$ work. There are i^2 pairs of invocations for which this query must be performed, so the total work is $O(i^2 log(i))$.[1]

Each time the two queried invocations are found to race, an insertion into *FunctionRaces* is performed. This insertion can happen $O(i^2)$ times, but the structure only grows to size $O(f^2)$ (if keeping invocation pairs in *FunctionRaces*, they are only appended to lists for their associated function pairs, not kept as elements in the top-level, sorted data structure). Total work for the insertions is $O(i^2 log(f))$.

The data structure *Causality* is an interim result used to build *FunctionRaces*. Combining the cost of constructing *Causality* with the costs to construct *FunctionRaces*, the total work in building the *FunctionRaces* data structure is $O(mc \cdot log(m) + ci^2 log(i) + i^2 log(i) + i^2 log(f))$. Since $f \leq i$, this expression reduces to $O(mc \cdot log(m) + ci^2 log(i))$.

UseDef

Construction of *UseDef* requires static analysis of every system function to determine its *use*() and *def*() sets. If s is the size of a function's text, this computation is $O(f(s + x \cdot log(fx)))$, where s is the time to perform the static analysis, and $x \cdot log(fx)$ is the time to insert entries into *UseDef*.

Detection Algorithm

There are $O(f^2 x)$ iterations in the nested loops described in section 5.3.2, and the *UseDef* lookup in the inner loop can be performed in $O(log(fx))$. So the complexity

[1] actually, results for the same function pair could be cached, such that *Causality* is only queried $O(f^2)$ times and the cache is queried $O(i^2)$ times. The cache only has size $O(f^2)$, so the total work could be reduced to $O(f^2 log(i) + i^2 log(f))$.

of this work is $O(f^2x \cdot log(fx))$.

Combining the work to build *FunctionRaces* and *UseDef* with this work to detect the destructive races, we get a total runtime cost of $O(mc \cdot log(m) + ci^2 log(i) + x \cdot log(fx) + f^2x \cdot log(fx))$. The third term is smaller than the fourth, so this can be simplified to $O(mc \cdot log(m) + ci^2 log(i) + f^2x \cdot log(fx))$.

5.4 Message Competition

I described in section 5.3.1 how to create a causality graph for a system execution E. This graph represents causality between pairs of invocations, pairs of messages, and pairs with one invocation and one message. To detect message competition we also must consider the messages' types. Each function in the system definition declares a list of inputs that function consumes, and each input specifies a message type to which that input can be bound.

If some function f declares 2 inputs of types t_1 and t_2, then there must exist messages m_1 and m_2 of the appropriate types, and they must coexist in order for f to invoke. If m_1 and m_2 exist in E and they are not causally ordered (i.e., $\neg(m_1 \xrightarrow{hb} m_2) \wedge \neg(m_2 \xrightarrow{hb} m_1)$), then it is possible that f was *enabled* with the binding $< m_1, m_2 >$ in execution E.

5.4.1 Ordering within message types

The message competition algorithm I present here partitions the system's messages by type, and then analyzes causality looking for feasible bindings. First, I will show that the coexistence of two messages of the same type is a sufficient condition for message competition. If 2 messages of the same type race and either one of them is consumed, then the execution contains message competition.

Lemma 5.4.1 *In execution E, if 2 messages of the same type race, and at least one of them is consumed, then E contains message competition.*
proof: Suppose two messages, m_1 and m_1' are of the same type, they race, and m_1 is consumed along with $\{m_2, ..., m_k\}$ by an invocation i. For proof by contradiction, suppose that there is no message competition.

If m'_1 races with all of $\{m_2, ..., m_k\}$, then there is clearly competition, because $< m'_1, m_2, ..., m_k >$ is a feasible binding for the same function of which i is an invocation. So there must be some $m_i \in \{m_2, ..., m_k\}$ for which either:

1. $m_i \xrightarrow{hb} m'_1$, or
2. $m'_1 \xrightarrow{hb} m_i$

1 is not possible. Since m_i is consumed with m_1, $m_i \xrightarrow{hb} m'_1$ would imply that $m_1 \xrightarrow{hb} m'_1$, but we know that m'_1 and m_1 race. If 2 holds (i.e., $m'_1 \xrightarrow{hb} m_i$), then m'_1 must be consumed in order to produce a future containing m_i. I name the consuming invocation i' and the other messages consumed with m'_1 as $\{m'_2, ..., m'_j\}$, As before, since we have pre-supposed no message competition exists, there must be some $m'_i \in \{m'_2, ..., m'_j\}$ for which:

2.1 $m'_i \xrightarrow{hb} m_1$, or
2.2 $m_1 \xrightarrow{hb} m'_i$

Since m'_i is consumed with m'_1, 2.1 implies that $m'_1 \xrightarrow{hb} m_1$, which is false. Now consider the combination of 2.2 and 2, above. Since m'_i and m'_1 are consumed together, case 2 above ($m'_1 \xrightarrow{hb} m_i$) implies that $m'_i \xrightarrow{hb} m_i$. Combine this with 2.2 ($m_1 \xrightarrow{hb} m'_i$) and the transitivity of $\xrightarrow{hb}$ implies that $m_1 \xrightarrow{hb} m_i$. But m_1 and m_i are consumed together by i, so they race: a contradiction. Cases 1, 2.1, and 2.2 are all impossible. ■

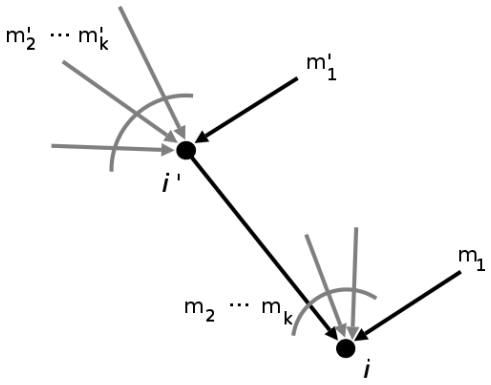

Figure 5.4.1: Lemma 5.4.1, case 2

Note that if a message is never consumed, it does not happen before any other messages. Combining this fact with the requirement that all consumed messages of the same type be causally ordered (lemma 5.4.1), it is possible to sort messages of the same type in a list, with any unconsumed messages at the bottom, arbitrarily ordered between each other. If the sort fails (i.e., some pair of messages (m_1, m_2) exists where $\neg(m_1 \xrightarrow{hb} m_2) \wedge \neg(m_2 \xrightarrow{hb} m_1)$ and at least one of them is consumed in E), then competition is detected (lemma 5.4.1). The list containing messages of type t is named M_t.

5.4.2 Message Competition Properties

Even if all messages of the same type are causally sorted, message competition can exist. For example, suppose that two functions $foo()$ and $bar()$ each consume a single message of type t. The system in which these are included is destined to have message competition. As soon as a message of this type is produced, two bindings including this message are enabled. Each contains only this single message, but one is bound to the function $foo()$ and one to $bar()$. Any two functions that specify the same message types for their inputs will interfere with one another in this manner.

As another example, consider two functions, again named $foo()$ and $bar()$. This time $foo()$ consumes a single message of type t, and $bar()$ consumes two messages, of types t and t'. An execution in this system will contain competition if it ever produces two racing messages m and m' of types t and t', respectively.

$foo()$ and $bar()$ compete for m, because m might be consumed by $foo()$ in a binding by itself, or by $bar()$ in a binding that includes both m and m'. This example, too, is a special case, because the input set for $foo()$ is a subset of the input set for $bar()$, which means that the existence of a feasible binding for $bar()$ implies that there is message competition.

In general, message competition is a possibility between any two functions with a common input message type. In figure 5.4.2, there is competition for message m_2 between two feasible bindings. The first binding, with m_1 and m_2 is realized in the execution, and leads to an invocation i. The second binding, with m_2 and m_3, is *not* realized. Due to the competition, the invocation i' that would result if this binding were realized cannot occur. Invocation i consumes m_2, so m_2 is no longer available for i'.

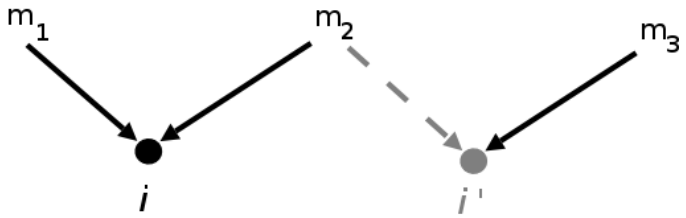

Figure 5.4.2: Competition for message m_2

Note that the functions associated with the two feasible bindings in figure 5.4.2 could be the same one. If so, that would mean that m_1 and m_3 have the same type.

5.4.3 Message Competition Detection

In this section, I present an algorithm that detects whether an execution E contains message competition. When competition is present, the algorithm returns the first feasible binding that was not realized in E. There must be competition for at least one message in this returned binding, because otherwise the binding would be realized.

If the sorting of same-type messages succeeds (section 5.4.1), then the detection algorithm presented here (in figures 5.4.3 and 5.4.4) tries every function in the system, searching for a feasible binding that was not realized in E. If one is found, then message competition is detected. If none are found, then the system is free of message competition.

The top-level function $findCompetition()$ iterates over the set of functions in the system definition. For each function, $findCompetition()$ retrieves the sorted list of messages M_t for each message type t specified in the function's inputs and stores them in an array of lists, $candidates[\,]$: each position in the array holds the list of messages for one input type. It then passes this array into $findCompetingBinding()$, which searches through the lists.

The $findCompetingBinding()$ implementation shown in figure 5.4.4 is simplified by assuming that all messages of the same type are ordered (i.e., there are no unconsumed messages that race with one another). It looks at pairs of messages,

```
    bool findCompetition() {

1       Set<Function> functions = allFunctions();
2       for func (functions) {                          // O(f)
3           List<Msg>[] candidates = new List<Msg>[func.inputs.length];
4           foreach i (candidates.length) {             // O(c)
5               candidates[i] = msgsOfType(typeof(func.inputs[i]));
                                // returns sorted list - O(log(t))
            }

6           List<Msg> binding = findCompetingBinding(func, candidates);
        }

7       return (binding != null);
    }
```

Figure 5.4.3: findCompetition()

each pair consisting of the first message of two input types (at the tops of the lists in *candidates*[ ]). If causality is found on line 7 ($m_i \xrightarrow{hb} m_j$), then the transitivity of $\xrightarrow{hb}$ implies that m_i happens before every message in the list *candidates*[j] (i.e., those of the same type as m_j). That implies that no bindings can be found that contain m_i, so m_i can be removed from consideration (lines 8 and 10-11).

If the causality check on line 7 does not hold for any pair (m_i, m_j), then *removals* is empty after the outer *for* loop terminates (line 3). In this case, a binding has been found containing all the messages at the tops of the lists (line 12). If the binding b is not one that was realized in E, then message competition has been detected, and *findCompetingBinding*() simply returns b.

Checking if the binding was realized in E is straightforward. For the first message in b, simply lookup the binding and associated function by which this message was actually consumed in E. Compare that function with *func* and each input message to the corresponding one in b. If the function is different, or any input contains a different message, then b was not realized. If b *was* realized, then the messages in b were consumed together, so they all have the same causal future. Thus, all the messages in b happen before all the other messages in the lists, so they

```
List<Msg> findCompetingBinding(Function func, List<Msg>[] candidates) {

    // candidates: for a given function with c inputs, this is an
    //   array of size c, where each element is a list of messages
    //   of the proper type

1   while (! hasEmptyList(candidates)) {
2       List<int> removals;
3       for int i ( 0 .. candidates.length-1 ) {
4           for int j ( 0 .. candidates.length-1 ) {
5               Msg mi = candidates[i].top();
6               Msg mj = candidates[j].top();
7               if (mi != mj && happensBefore(mi,mj))  {
8                   removals.add(i);
                }
            }
        }

9       if (! removals.empty()) {
            // remove messages that cannot be part of a binding
10          for int i (removals) {                  // O(c)
11              candidates[i].pop();
            }
        } else {
            // found a binding
12          Binding b = candidates.popAllTops();    // O(c)
13          if (! wasRealized(func, b)) {           // O(c)
14              return b;
            }
        }
    }
15  return null;
}
```

Figure 5.4.4: findCompetingBinding()

can be removed all at once (line 12).

The loop terminates if an unrealized binding is found (line 14), or when $hasEmptyList(candidates)$ becomes true. This function call checks if there is any list in $candidates[\ ]$ with no messages left. If the loop terminates without finding a binding, $findCompetingBinding()$ returns $null$, and $findCompetition()$ returns false, indicating there is no message competition.

5.4.4 Dropping the simplifying assumption

If we drop the simplifying assumption that all messages of the same type are ordered, it looks bad at first. If, in some iteration, a binding is not found, but some list in $candidates[\ ]$ contains unsorted messages at the top, it is hard to safely remove any messages. If it is not the case that $m_j \xrightarrow{hb} m'_j$ for every other message m'_j in $candidates[j]$, then the condition $m_i \xrightarrow{hb} m_j$ is not sufficient to know that it is safe to remove m_i.

At the very least, before discarding m_i we may have to compare it to every message in $candidates[j]$. But in fact that approach will not work either, because there might be some m_i that races with one message in $candidates[j]$ and some other m_k (of a different type) that races with a different message from $candidates[j]$. Then we have to consider both possible futures (removing m_i and m_k separately). Analysis along this path degrades into an algorithm that is exponentially expensive in the general case.

But the problem can be solved by reversing the lists. Sort messages in each list in $candidates[\ ]$ in reverse order, with the unconsumed, intra-type-racing messages at the top (in arbitrary order relative to each other). With the lists reverse sorted, messages on the *other* side of the $\xrightarrow{hb}$ relationships are removed from the lists (i.e., if $m_i \xrightarrow{hb} m_j$, remove m_j). The only change to $findCompetingBinding()$ is on line 8, shown here with some context:

```
7               if (mi != mj && happensBefore(mi,mj))  {
8                   removals.add(j);
                }
```

Now, an iteration will always produce either a binding or a set of messages to safely remove. To see this, note that if $m_i \xrightarrow{hb} m_j$, then m_i cannot be one of the unconsumed messages at the top of a list. These messages do not have causal

futures, so they can never play the role of m_i when $happensBefore(m_i, m_j)$ returns *true* on line 7. And since m_i cannot be one of these unconsumed messages, the list $candidates[i]$ contains only causally ordered messages, sorted in reverse order. So $m_i \xrightarrow{hb} m_j$ does indeed imply that *all* messages in $candidates[i] \xrightarrow{hb} m_j$, and it is therefore safe to discard m_j.

5.4.5 Message Competition Detection Complexity

The variables in the complexity analysis that follows have the following definitions:

m number of messages in E
f number of functions in the system definition
c number of messages consumed by a function
v number of messages of a given type
t number of message types ($t = \frac{m}{v}$)

I assume that we already have a list of the system's functions, accessed in $findCompetition()$ (figure 5.4.3) by the call to $allFunctions()$, and a data structure $Causality$ with complete causality information for execution E (see section 5.3.1). Using the $Causality$ data structure, the query $happensBefore(m_i, m_j)$ called in $findCompetingBinding()$ (in figure 5.4.4), can be implemented with a run-time complexity of $O(log(i))$.

In detecting message competition, the first computation performed is to sort each list of same-type messages. Each sort computation can be performed by making $O(v \cdot log(v))$ comparisons and swaps, and each comparison is a query to $Causality$ that takes $O(log(i))$ time. There are $t = m/v$ lists of messages to sort, so the sorts can be performed in $O(\frac{m}{v} \cdot v \cdot log(v) log(i))$, which can be simplified to $O(m \cdot log(v) log(i))$. If we store each sorted list by their type in an efficient data structure, retrieving the list for a given type is $O(log(t))$.

Once the lists are sorted, the top-level function $findCompetition()$ (figure 5.4.3) iterates through the system's functions and retrieves the relevant sorted lists for all of each function's inputs. These lookups take $O(c \cdot log(t))$ in each iteration. After retrieving message lists for a given function, $findCompetition()$ calls $findCompetingBinding()$.

$findCompetingBinding()$ (figure 5.4.4) has an outer *while* loop in which the first action in each iteration is to check if any list of messages is empty by calling

hasEmptyList(*candidates*). This function could be made to operate in constant time by storing the answer in a *boolean* and updating it to *true* the first time removal from a list makes that list empty.

The doubly-nested *for* loops (figure 5.4.4, lines 3-4) affect c^2 iterations in which the causality relationship between two messages is queried each time. This query is $O(log(i))$, so the complexity of the loops is $O(c^2 log(i))$. Adding an element to a list (line 8) is $O(1)$, so this work does not contribute to the complexity of the loops.

After the loops, the remaining work is less. $findCompetingBinding()$ performs up to $c - 1$ removals from the lists in lines 10-11, and each is $O(1)$. This term $O(c)$ is less than the complexity of the loops, so it does not contribute to the time complexity for $findCompetingBinding()$. Alternatively, if $removals$ is empty, $findCompetingBinding()$ performs the work in lines 12-14. On line 12, it removes exactly c elements from the lists, again each removal requiring constant execution time.

As I described in section 5.4.3, the call to *wasRealized*() performs up to $c + 1$ simple comparisons: one for the function and c for the messages. Again, this $O(c)$ computation does not contribute to the overall complexity. Thus, the complexity of a single iteration of the *while* loop in $findCompetingBinding()$ is $O(c^2 log(i))$. At least one message will be removed in each iteration of the *while* loop that does not detect message competition. Therefore, the loop iterates at most $O(cv)$ times (the total number of messages in the lists). Thus, the overall complexity of $findCompetingBinding()$ is at most $O(vc^3 log(i))$, if we can only remove a single message from the lists in each iteration.

The runtime complexity of $findCompetingBinding()$ is greater than that of the other work $findCompetition()$ performs (gathering the sorted lists in $O(c \cdot log(t))$). Thus, the overall complexity of $findCompetition()$ is $O(fvc^3 log(i))$. Added to the work of sorting the lists of same-type messages, the total work for message competition detection is $O(m \cdot log(v) log(i) + fvc^3 log(i))$.

Chapter 6

Localized Nondeterminism

Suppose that the system contains a race condition. Then more than one invocation can define some system variable x for an invocation i, and the system is nondeterministic. While strictly nondeterministic, it is possible for the end result of the system's computation to remain deterministic. For example, consider the following possibilities.

Commutative predecessors It is possible that x holds the same value upon execution of i even if it may have been defined by more than one writer invocation. In the case that every writer that potentially defines x for i writes the same value for x, as long as these writers are deterministic, x will have that same value every time i uses it. Another possibility is that the racing writers perform commutative operations, and each such writer i_{dx} satisfies $i_{dx} \xrightarrow{hb} i$. Then, again, i will use the same value every time.

Close enough This case is similar to the previous one, except that the value i uses for x might not hold the exact same value across multiple runs. i's predecessors might race and produce different values for x, but every possible value that i might use for x contributes the same to the expression(s) in which it appears. For example, x might have values 3, 4, or 18, depending upon the outcome of earlier races, but the only expression in which x appears in i is $(x > 1)$, which evaluates to true for all these possible values. If all uses of system variable x are similar until such time as x is assigned a new value, then the system still behaves deterministically.

6.1 Localized nondeterminism

Another possibility is that a race condition truly causes non-deterministic behavior in i, but the nondeterminism introduced to the system is localized. For example, consider two threads of control that perform lazy initialization of some shared entity, such as an encryption cipher. When created, a cipher often must perform expensive computation, such as generating a randomized encryption key. It also might need to parse a config file to determine encryption parameters or initialize internal data structures. These startup costs make the cipher a good candidate for both lazy initialization and sharing between threads and across execution time. Initialize it once and then re-use it multiple times.

A typical lazy initialization implementation might look something like that in figure 6.1.1, where `createCipher()` performs the expensive computation(s). If two invocations that race call `getCipher()`, each one may or may not write to the shared variable `cipher` $\in X$. In fact, they may both call `createCipher()` and set `cipher`. But in the end `cipher` is initialized to an equivalent `Cipher` object, and the calling threads are unaffected by the race condition.

```
Cipher cipher = null;

Cipher getCipher() {

    if (null == cipher) {
        cipher = createCipher();
    }
    return cipher;
}
```

Figure 6.1.1: Lazy initialization

One approach to identifying localized nondeterminism is to identify common types of localized nondeterministic behavior, such as well-known commutative operations, load-balancing producer/consumer schemes, and optimistic look-ahead computations. In this section, I pursue a more general approach. I postulate that it is interesting to consider a class of nondeterministic behaviors that are short-lived.

In other words, once a race condition is identified, convergence to a deterministic system state is expected to occur within some short distance along the causality paths.

6.1.1 Subgraphs

A system execution can be represented with a directed, acyclic graph, where the nodes are invocations and the directed arcs are the messages.

Definition 6.1.1 - execution graph
The execution graph G for a system execution E is a DAG composed of:

nodes *the set I of all invocations in E*

arcs *the set M of messages in E*

$\mathbf{s_0}$ *a source node* s_0

A message produced by invocation i and consumed by i' is modeled as a directed arc from source node i to sink node i' (see figure 6.1.2). Messages in M_0 have node s_0 as their source node. The $\xrightarrow{hb}$ relation is equivalent to reachability: if $i \xrightarrow{hb} i'$ then there is a path in G from i leading to i', and a path from i to i' implies that $i \xrightarrow{hb} i'$. A path that starts at s_0 and terminates at an invocation that produces no output messages is called a *fullpath*.

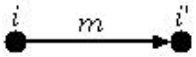

Figure 6.1.2: Invocation i produces m consumed by i'. *i happens before* i'.

To identify a portion of a system execution in which nondeterminism is contained, I will use a *subgraph* construct, defined here.

Definition 6.1.2 - subgraph
A set of invocations S in the execution graph G form a subgraph iff:

- *they are connected*
- $\forall i \in G, i_{s1}, i_{s2} \in S :: i_{s1} \xrightarrow{hb} i \xrightarrow{hb} i_{s2} \Rightarrow i \in S.$

Definition 6.1.2 expresses that for S to be a *subgraph* of G, any invocation i that is strongly connected to S *must be in* S. So in the graph in Figure 6.1.3, if we wish to include i_1 and i_2 in a subgraph, i_3 must also be included. Lemma 6.1.3 follows trivially from the subgraph definition.

Lemma 6.1.3 *Given a set of invocations S that contains i_S and i'_S, and an invocation $i \notin S$ for which:*

$$i_S \xrightarrow{hb} i \xrightarrow{hb} i'_S$$

i is included in **all** *subgraphs that contain S.*

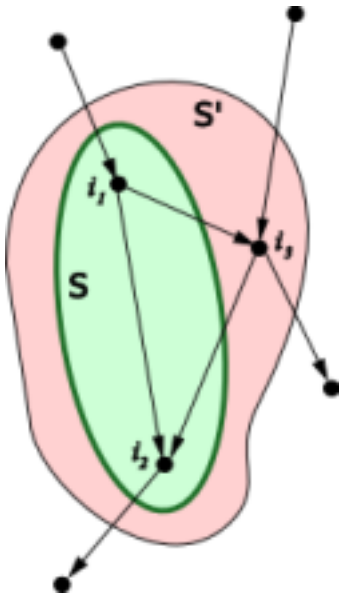

Figure 6.1.3: S' is a subgraph. S is not.

The following definitions overload the $\xrightarrow{hb}$ relation and *svc* to include subgraphs. Note that with these additions, the meaning of *races with* and *races destructively* are extended as well.

Definition 6.1.4 $i \xrightarrow{hb} S \quad \stackrel{def}{=} \quad i \notin S \ \wedge \ \langle \exists i_S \in S :: i \xrightarrow{hb} i_S \rangle$

Definition 6.1.5 $S \xrightarrow{hb} i \quad \stackrel{def}{=} \quad i \notin S \ \wedge \ \langle \exists i_S \in S :: i_S \xrightarrow{hb} i \rangle$

Definition 6.1.6 $S_1 \xrightarrow{hb} S_2 \quad \stackrel{def}{=} \quad S_1 \cap S_2 = \emptyset \ \wedge \ \langle \exists i_{S1} \in S1, i_{S2} \in S2 :: i_{S1} \xrightarrow{hb} i_{S2} \rangle$

Due to the properties of a subgraph (definition 6.1.2), $\xrightarrow{hb}$ is still antireflexive and transitive. However, it is no longer asymmetric, since it is possible for $S_1 \xrightarrow{hb} S_2$ and $S_2 \xrightarrow{hb} S_1$. In this section, I avoid this relationship by choosing carefully how to define subgraphs in an execution graph.

To prevent interference between a subgraph and the rest of the system, its invocations must be *isolated*, as described in the following definition.

Definition 6.1.7 - isolated subgraph
A subgraph S is an isolated subgraph iff no invocation external to S races destructively with S.

If a subgraph S is not an isolated subgraph, we can attempt to identify an isolated subgraph by adding to S invocations that race destructively with S. As stated in lemma 6.1.8, the invocation that we add must be included in *any* isolated subgraph that contains the invocations in S, so an aggressive algorithm for growing S can be used to find a minimal isolated subgraph. By continually adding to S invocations that race destructively with S and any invocations for which $S \xrightarrow{hb} i \xrightarrow{hb} S$ (so that S remains a subgraph), we can hopefully find an isolated subgraph. If not, the system contains non-localized nondeterminism, and cannot be analyzed.

Lemma 6.1.8 *Given a set T of invocations, and invocation i that races destructively with any invocation in T, i is included in* **all** *isolated subgraphs that contain T.*

6.1.2 Nondeterministic seeds

When an invocation i consumes messages produced by deterministic invocations, but there is no sole writer for some $x \in use(i)$, I call i a **seed invocation**, reflecting that it is a seed of nondeterminism. I showed in section 4.4 that missing sole writers are the direct result of destructive race conditions. As an example, suppose that i races with another invocation i' for which $x \in def(i')$. Depending on which invocation runs first, i might use the x value defined by i', or the value x holds before i' changes it. Informally, a seed invocation is the start of nondeterminism. Its nondeterministic

behavior can change the behavior of other invocations that either exist concurrently or follow it.

Now suppose that i not only uses x, but also defines x. And suppose that i' also uses x. In this case, i' is also a nondeterministic seed invocation. In cases like this, where nondeterministic behavior is seeded by multiple invocations that interact, I wish to analyze them together. From a single seed invocation, I define a **seed subgraph**. For brevity, I refer to the seed subgraph, defined below, simply as the **seed**.

Definition 6.1.9 - seed subgraph
Given a seed invocation i, its associated seed subgraph is the minimal isolated subgraph containing i.

Note that in the example I gave above, where i and i' are both seed invocations, since the seed subgraph for i is an isolated subgraph, it must include i'. Similarly, the seed subgraph for i' must include i.

In general, not all invocations that exist concurrently with or after the seed are affected by the nondeterminism it introduces. Ignoring for now the effect of message competition, the seed subgraph S also cannot directly affect the behavior of any invocation i that races with S if there is no variable conflict between them (i.e., $\neg svc(i, S)$). It can, however, do so indirectly by changing the behavior of some i' for which $svc(i', S) \wedge svc(i', i)$ (for example).

6.1.3 Boundaries

Given an acyclic directed execution graph G, consider a pair of cuts across the graph, each a set of messages. They form a **boundary** if the following all hold for every fullpath f that crosses either cut:

1. f crosses both cuts
2. f crosses each cut exactly once
3. f crosses the input cut before the output cut

In a fullpath that crosses the boundary's cuts, the message from the input cut contained in the path is a boundary **input message**, m_{in}. The message from the output cut contained in the path is an **output message**, m_{out}. Any invocation

i that occurs on the fullpath either before the input cut or after the output cut is **outside** the boundary (i.e., $i \xrightarrow{hb} m_{in} \vee m_{out} \xrightarrow{hb} i$). If, on the other hand, i occurs between the two cuts (i.e., $m_{in} \xrightarrow{hb} i \xrightarrow{hb} m_{out}$), i is **inside** the boundary.

To constrain the effects of nondeterminism introduced by a seed, I want to identify a boundary that contains the seed and the nondeterministic execution that follows. If defined and *verified* properly, the boundary will maintain its external form across all feasible executions, as long as its inputs remain the same. In this way, the nondeterministic execution contained within the boundary behaves similarly to a deterministic function.

The inputs to the bounded nondeterminism include the boundary's input messages and the set of variables read by invocations in the boundary. If the nondeterministic nature of computation in the boundary causes a shared variable to be read in some executions but not in others, that variable is still an input to the boundary.

Of course, since the boundary execution is nondeterministic, it might be defining and subsequently using some of its input variables, and it might define different values in alternate executions. But in verifying the boundary execution in isolation, we will verify that these conflicts resolve themselves deterministically. The key is that such nondeterministic influences cannot be external. For example, if an external invocation writes to a shared variable that is one of the boundary's inputs while the boundary execution is in progress, then even though it is *verified* (definition 6.1.10), it might not converge to a deterministic end.

The boundary's outputs include its consumption and production of messages, the definitions of shared variables it performs, and the $\xrightarrow{hb}$ relationships it must maintain between alternate executions. As defined formally in definition 6.1.10, a boundary is a **verified boundary** if it has been shown that given the same inputs, it provides the same outputs, regardless of the outcomes of internal race conditions.

Definition 6.1.10 - verified boundary
For a given set of inputs, the nondeterministic execution within a boundary has been **verified** *with respect to a reference execution E_0 iff it has been shown to produce the same outputs for all feasible sub-executions (i.e., local executions involving the boundary's input and output messages and all invocations and messages inside the boundary).*

The boundary's input conditions are defined to be as they are in the reference execution E_0:

- *boundary input messages*
  - *all input messages in B_{in} exist*
  - *input messages in B_{in} contain the same data and data values*
  - *no additional messages are produced outside the boundary that do not exist in E_0 and that can cause message competition involving messages from the boundary execution*
- *shared variables*
  - *when an invocation inside the boundary uses x, it should hold either the verified input value from E_0, defined by the boundary's sole writer for x, or a value defined by another invocation inside the boundary*

If the inputs described above are as they are in E_0, it is guaranteed to produce deterministic outputs as follows:

1. *output messages*

   the same set, no more, no less, with the same data and values

2. *shared variables*

   the same shared variable use/def sets

3. *input message consumption*

   (a) *the execution consumes all $m_{in} \in B_{in}$*
   (b) *the execution does not consume any message outside the boundary that existed in E_0*
   (c) *the execution does not compete for any message outside the boundary that existed in E_0*

4. *causality*

   (a) *the same $\xrightarrow{hb}$ relationships between input messages and output messages*
   (b) *the same $\xrightarrow{hb}$ relationships between x definitions (i.e., all invocations in any execution that define x) and the messages in B_{in} and B_{out}*

(c) the same $\xrightarrow{hb}$ *relationships between x uses and the messages in* B_{in} *and* B_{out}

In the definition above, item 3b has subtle meaning. If a local execution consumes a message that is neither a member of B_{in} nor a message produced internally in B, or if there is competition for a message that existed in E_0, then the boundary is not a verified boundary. Note that if a subsequent global execution produces a message that did not exist in E_0, that is a violation of the input pre-conditions, not an indication that the boundary is not verified.

6.1.4 Futures

Given a reference system execution E_0, a seed subgraph S, and a boundary B that contains S, I need to identify in an alternate execution E the invocations in the boundary computation. They must converge to a deterministic end for the boundary to form in E. These are the invocations that causally follow the messages in B_{in}, the boundary's input cut.

Definition 6.1.11 - future, F
For an execution E *and a set of messages* M*, the associated future* F *contains an invocation* i *if and only if* $\langle \exists m \in M :: m \xrightarrow{hb} i \rangle$.

Thus, the invocations I wish to identify are those in the input cut's (B_{in}) **future** in E. Note that there is no guarantee that all of the messages in B_{in} actually exist in E. In this case, we will be interested in the future for the subset of B_{in} that *does* exist in E. If none of the messages in B_{in} exist in E, then the boundary and associated seed do not exist in E.

Definition 6.1.12 - converging future
For a verified boundary B *and system execution* E*, an associated future* F *is a* **converging future** *(*CF*), iff it* converges *to* B_{out}*, the boundary's output cut, and preserves all of* B*'s outputs as prescribed when* B *was verified (definition 6.1.10).*

If F does not converge, it is a **non-converging future** (NCF), and it indicates that some input to B is different than that for which B is verified.

In the following definition, when collapsing sets of invocations into single nodes, arcs between two invocations in the collapsed set are swallowed into the collapsed node. Arcs that are connected only on one end to an invocation in the collapsed set are maintained, connected in the resulting graph to the collapsed node on that end.

Definition 6.1.13 - converging system
Given a system and reference execution E_0, suppose every seed in E_0 is enclosed in a boundary that exists in all feasible system executions. For every such boundary B and every feasible execution E, collapse B into a single node in the execution graphs for E_0 and E. From the original system execution graphs G_0 and G, this action produces graphs G'_0 and G'.

If the following all hold, then the system is a **converging system***:*

- *G' and G'_0 are the same graph (i.e., they contain all the same nodes and connecting arcs)*
- *each boundary node is verified for its inputs in E_0, and produces the verified outputs in E (definition 6.1.10)*
- *each invocation that is a node in G' and G'_0 defines the system's shared variables the same*

A converging system is the goal. In the next section I show that if boundaries are identified and verified for all seeds, and if all invocations and boundaries contain sole writers for each of their inputs, then the system is a converging system.

6.1.5 Sufficient Conditions for Convergence

In proving the convergence of systems with verified boundaries, I want to treat the execution of a boundary-contained computation as a single step in the induction in theorem 4.3.1. In reality, the invocations in the boundary may be interleaved with external invocations. But I can transform the execution into an equivalent one in which the internal invocations are all together.

Consider each pair (i, i') consisting of an internal invocation i and an external invocation i' from E. If the boundary is verified in E_0, and if external invocations and boundaries are deterministic in E, then i and i' must either be causally ordered, or they are independent, just as in E_0.

And if they are independent, then reordering the interleaved invocations such that those internal to the boundary are contiguous produces an equivalent execution (as in [64]). Thus, I can optimistically treat the boundary execution as an atomic event in my induction. If the full induction over E produces a causality graph identical to E_0 (which it does, as proven in 4.3.1), then this simplification does not affect the analysis of any of the inductive step.

And now, the induction proof is identical to that in theorem 4.3.1. When a boundary executes, it satisfies the induction hypothesis by defining the same shared variable values and producing the same messages as in E_0.

Theorem 6.1.14 *If one of the following holds for every invocation i and $x \in use(i)$ in a feasible execution E_0:*

- *i uses x's initial value (i.e., $\langle \forall i_{dx}, x \in def(i_{dx}) :: i \xrightarrow{hb} i_{dx} \rangle$), or*
- *i has a sole writer for x, or*
- *i is in a verified boundary B*

and if for every verified boundary B and $x \in use(B)$:

- *B uses x's initial value (i.e., $\langle \forall i_{dx}, x \in def(i_{dx}) :: B_x \xrightarrow{hb} i_{dx} \rangle$), or*
- *B has a sole writer for x*

and if there is no message competition in E_0 or resulting from alternate sub-executions in a boundary, then the system converges.

proof: *By induction*

For an arbitrary execution E of the same system, I show that $E \equiv E_0$.

Induction hypothesis - if the following conditions hold for prefix(E, l), then they hold for prefix$(E, l{+}1)$:

1. prefix(E, l) is a consistent cut from E_0
2. prefix(E, l) contains no invocations or messages that did not exist in E_0
3. every invocation $i \in$prefix(E, l) defines the same values to the same shared variables as in E_0

4. every invocation $i \in$prefix(E, l) produces the same messages with the same payloads as in E_0

base case: $l = 0$ - trivially satisifed

step: Analysis is identical to that in theorem 4.3.1. Both single invocations and verified boundary computations maintain the induction hypothesis if their inputs are consistent with those in E_0, and the induction hypothesis ensures that they are. ■

Chapter 7

Boundary Detection

7.1 Local Execution

In this section I describe a process using re-execution to explore the state space within a boundary.

7.1.1 Enabled Bindings

To find the end of localized nondeterminism, we must perform re-execution to explore the state space it realizes. The simplest approach is to execute all possible paths starting with the messages in B_{in} and the invocations in B that consume those messages. At each step in the re-execution, we need to know what invocations (and associated functions) are enabled. This is also a requirement of the underlying runtime system during normal execution.

As controlled execution proceeds, track the set of available messages, and keep them in a data structure named A. To determine the set of enabled bindings, consider each message $m \in A$. Lookup functions that consume a message of type $typeof(m)$. For each function f, search messages in A to satisfy the remainder of its input specification. I break this algorithm into two pieces. The first is $findEnabledBindings()$, shown in figure 7.1.1, which iterates through all the system's functions, calling the recursive subroutine $findBindings()$ on each, seeded with the empty Binding b. $findBindings()$ is shown in figure 7.1.2. It is responsible for the search through A for messages that match a single function's input specification (i.e., its signature).

```
1   Set<Msg> A = // available messages

2   bool findEnabledBindings() {

3       Set<Binding> bindings;
4       for func (getSystemFunctions()) {       // O(f)
5           Binding b;
6           bindings.add(findBindings(f,b,A));
        }
7       return bindings;
    }
```

Figure 7.1.1: findEnabledBindings()

```
1   void findBindings(Function f, Binding b,
                      Set<Msg> A, Set<Binding> bindings) {

2       if (b.len < f.input_spec.len) {
3           Set<Msg> candidates
                = msgsOfType(typeof(f.input_spec[b.len]));
4           for Msg cand (candidates) {
5               findBindings(f, b.append(c), A.remove(c), bindings);
            }
        } else {
6           bindings.add(b);
        }
    }
```

Figure 7.1.2: findBindings()

7.1.2 Complexity of $findEnabledBindings()$

The recursion in $findBindings()$ is $O(s)$ deep, where s is the size of a function's input specification. If there are multiple messages of the same type in A, then there is message competition, so assume there is typically only one. Retrieving it is $O(log(a))$, where a is the size of A. At line 5 in figure 7.1.2, creating a copy of b and A with candidate message c added and removed, respectively, is $O(s + a)$. But could also be done without creating copies in $O(log(s) + log(a))$. So the overall complexity for $findBindings()$ is $O(s(log(a) + log(s) + log(a)))$. And since $a \geq s$, this expression simplifies to $O(s \cdot log(a))$.

The $findEnabledBindings()$ routine in figure 7.1.1 calls $findBindings()$ in a loop $O(f)$ times (f is the number of functions in the system), so its overall complexity is $O(fs \cdot log(a))$.

7.1.3 Re-execution

To explore the possible paths of computation, we can use partial order reduction [100, 40] to prune the state space. In order to do so, I need to define a dependence relation to be used by the partial order reduction algorithms. The following definition of a dependence relation is from [64].

Definition 7.1.1 *Given the set T of transitions (i.e., invocations), the set $\mathcal{D} \subseteq T \times T$ is a* **dependence relation** *iff for all $t_1, t_2 \in T$, $(t_1, t_2) \notin \mathcal{D}$ implies that:*

1. *if t_1 is enabled in state s and $s \rightarrow s'$, then t_2 is enabled in s iff t_2 is enabled in s'*
2. *if t_1 and t_2 are enabled in state s, then there is a state s' such that $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$.*

The first condition states that the occurrence of t_1 cannot enable or disable t_2. The second states that t_1 and t_2 are commutative.

In my system model, invocation i_1 enables another invocation i_2 if it produces a message that i_2 consumes. It is also possible for i_1 to disable i_2 if the two invocations compete for a common message. $\mathcal{D}$ must contain all such invocation pairs. I conservatively approximate commutativity of invocations by their shared variable independence. Thus, i_1 races destructively with $i_2 \Rightarrow (i_1, i_2) \in \mathcal{D}$. Note

that when performing re-execution, I have to use static analysis of the functions to determine whether two of them have a variable conflict, rather than actually observing their behavior at runtime as I do elsewhere. This is consistent with others' approaches [40].

Re-execution of a given computation, or trace, terminates when the end of the local nondeterminism is reached. This is occurs when B_{out} is reached. To find the messages that comprise B_{out}, I assume the existence of an oracle that identifies each message in B_{out}. In future versions of this research, an automated detection algorithm might take the form of heuristics that use message types and analysis of the $\xrightarrow{hb}$ relationships of messages relative to those in B_{in}. It also might employ assistance from a human.

With the messages in B_{out} identified, I continue controlled execution in each trace until all the messages in B_{out} are produced, or until progress requires consumption of one of these messages. In the latter case, a boundary cannot be formed, and the system is not a converging system.

7.2 Boundary Verification

For a given local execution BE, once B_{out} is identified and the boundary defined, we must detect whether BE satisfies the requirements of a verified boundary. Copied here from the definition of a verified boundary (definition 6.1.10), the following are the outputs that we must verify for every trace in B:

1. output messages

   the same set, no more, no less, with the same data and values

2. shared variables

   the same shared variable use/def sets

3. input message consumption

   (a) the execution consumes all $m_{in} \in B_{in}$
   (b) the execution does not consume any message outside the boundary that existed in E_0
   (c) the execution does not compete for any message outside the boundary that existed in E_0

4. causality

   (a) the same $\xrightarrow{hb}$ relationships between input messages and output messages
   (b) the same $\xrightarrow{hb}$ relationships between x definitions (i.e., all invocations in any execution that define x) and the messages in B_{in} and B_{out}
   (c) the same $\xrightarrow{hb}$ relationships between x uses and the messages in B_{in} and B_{out}

The first two conditions are trivial to verify, and so are conditions 3a and 3b. In each case, simply observe the sets of messages and shared variables and their data in E_0, and verify them and check them off in each local execution. Detecting whether the necessary causality relationships are preserved (condition 4) and whether there is message competition between the local execution and the rest of the reference global execution E_0 (condition 3c) are the subjects of section 7.3.

7.3 Boundary Message Competition Detection

We must detect whether the local boundary execution BE creates message competition. In addition to competition within the boundary, the local execution BE might cause competition involving messages outside the boundary: those observed in E_0 that are not in B. BE can produce different causality relationships than BE_0 or even different messages, and as a result create competition that did not exist in E_0.

7.3.1 Causality

In section 5.4, I solved the problem of detecting message competition for a single execution E of the whole system. In that section I employ an execution graph and a *Causality* data structure. To perform message competition detection for the local execution BE, I must create these structures for BE. The procedure is the same for each as it was in sections 5.1 and 5.3.1.

But using the *Causality* structure created for BE we are only able to detect $\xrightarrow{hb}$ between invocations and messages internal to the boundary – those in BE. I will also need to be able to detect a $\xrightarrow{hb}$ relationship, or a lack thereof, between any given internal message and any external one. As I construct *Causality*, I will also keep separately a list of $\xrightarrow{hb}$ relationships between each internal message and the

messages in B_{in} and B_{out}. For every message $m \in BE$, calculate and store the list $B\_inputs(m)$ containing $\{m_{in} \in B_{in} : m_{in} \xrightarrow{hb} m\}$ and $B\_outputs(m)$ containing $\{m_{out} \in B_{out} : m \xrightarrow{hb} m_{out}\}$.

These lists allow us to query whether some internal message m happens before an external message m_e and vice versa. To determine if $m \xrightarrow{hb} m_e$, it is equivalent to check whether $\langle \exists m_{out} \in B\_outputs(m) : m_{out} \xrightarrow{hb} m_e \rangle$. And to determine whether $m_e \xrightarrow{hb} m$, it is equivalent to determine whether $\langle \exists m_{in} \in B\_inputs(m) : m_e \xrightarrow{hb} m_{in} \rangle$. These queries can be answered using the B_inputs and $B\_outputs$ lists and the *Causality* structure for E_0 that were constructed for race detection and message competition detection for E_0.

7.3.2 Sorting the M_t lists

As with the detection algorithm described in section 5.4, we first want to determine if all of the messages in BE of the same type are ordered (i.e., all $m_t \in BM_t$ for each type t). They must be ordered with respect to each other, and also with respect to the *external* messages (in M_t) of the same type.

The work to verify that the messages in BM_t are ordered relative to the external messages in M_t is performed in two steps: one performed once, and the second performed for each feasible local execution of B. The first step is to identify a message b_t, the last message in M_t that occurs *before* B; and a message a_t, the first message in M_t that occurs *after* B. Message b_t satisfies $\langle \exists m_{in} \in B_{in} :: b_t \xrightarrow{hb} m_{in} \rangle$, and for every other message $b'_t \in M_t$ that also satisfies that expression, $b'_t \xrightarrow{hb} b_t$.[1] a_t is the mirror of b_t. It satisfies $\langle \exists m_{out} \in B_{out} :: m_{out} \xrightarrow{hb} a_t \rangle$, and for every other message $a'_t \in M_t$ that also satisfies that expression, $a_t \xrightarrow{hb} a'_t$.

In general, finding b_t can require a search from the beginning of the list of messages in M_t (which was constructed and sorted during message competition detection for E_0 - see section 5.4). The goal of the search is to find the two consecutive messages $m_t, m'_t \in M_t$ for which m_t happens before some $m_{in} \in B_{in}$, and m'_t does not. m_t is b_t. Note that if BE_0 contains a message of type t, then we can find b_t more easily. The pair m_t, m'_t is made of the first type t message from BE_0 and the message in M_t that immediately precedes it.

[1]Note: b_t might not exist if: (1) there is message competition in E_0, or (2) all messages in M_t happen after B. Similarly, a_t might not exist.

Once we know b_t, finding a_t is straightforward. If BE_0 contains one or more messages of type t, then a_t is simply the first message in M_t that follows these. And if BE_0 does not contain any type t messages, then a_t immediately follows b_t in M_t.

With b_t and a_t identified, the second step is to verify that the messages in M_t and BM_t are ordered. It must be performed for each feasible local execution of B. For each type t, we first sort the messages in BM_t, as we did for E_0 in section 5.4, and then we splice BM_t into the larger list M_t (replacing any type t messages from BE_0). The result is a modified list, M'_t, containing the external messages from E_0 and the internal messages from BE.

7.3.3 Competition Detection

With these modified M'_t lists, we can run the message detection algorithm from section 5.4.3. To limit the work, we first remove from the message lists any message that precedes all the messages internal to B. For example, consider some $f \in F$ that has 4 inputs. Two of them, of types t_1 and t_2, have candidates from BE (if none of the input types have candidates from BE, then we do not need to analyze f). Now suppose that in the list M_{t1}, the first message from BE is min_{t1} (min_{t1} is the first message in BM_{t1}). And the first message in BM_{t2} is min_{t2}.

In each of the four lists, we can immediately remove every message m_{tn} for which $m_{tn} \xrightarrow{hb} min_{t1} \ \wedge \ m_{tn} \xrightarrow{hb} min_{t2}$. In one list, M_{tn}, we can consider each m_{tn} in the sorted order of the list, removing it if $m_{tn} \xrightarrow{hb} min_{t1} \wedge m_{tn} \xrightarrow{hb} min_{t2}$. We stop as soon as we find one that cannot be removed. Similarly, given max_{t1} and max_{t2}, the last messages in BM_{t1} and BM_{t2}, we can remove each m_{tn} for which $max_{t1} \xrightarrow{hb} m_{tn} \ \wedge \ max_{t2} \xrightarrow{hb} m_{tn}$. To do so, start from the bottoms of the lists and work up. With what is left in the M'_t lists, we conduct message competition detection, as we did in section 5.4.3.

7.3.4 Complexity

The variables in the complexity analysis that follows have the following definitions:

m number of messages in E
m_B number of messages in BE
i number of invocations in E
i_B number of invocations in BE

p size of the larger of B_{in} and B_{out}
f number of functions in the system definition
c number of input messages to a function
v number of messages of a given type in E
v_B number of messages of a given type in BE
t number of message types in E ($t = \frac{m}{v}$)
t_B number of message types in BE ($t_B = \frac{m_B}{v_B}$)

Causality

The work of constructing *Causality* is the same as it was for the overall execution E_0 (section 5.3.3), but on a smaller graph. The runtime complexity is $O(m_B c \cdot log(m_B) + ci_B^2 log(i_B))$. The lists B_inputs and $B\_outputs$ for each internal message can be maintained during the same traversal in which we construct *Causality*. These relationships are fewer than the complete set in *Causality*, and they are discovered similarly, so the extra work does not contribute to runtime complexity.

To query the $\xrightarrow{hb}$ relationship of two internal messages using *Causality* is $O(log(i_B))$. To query the relationship of an internal message m and an external one requires that we query for a relationship between the external message and every message in $B\_inputs(m)$ and $B\_outputs(m)$. Each individual query is in the larger *Causality* structure for E_0, so it is $O(log(i))$. And it might have to be performed for every message in B_{in} and B_{out}, so the whole internal-to-external query is $O(p \cdot log(i))$.

BM_t sorts and splicing

Following the same analysis as in section 5.4.5, the task of sorting the BM_t lists has complexity $O(m_B log(v_B) log(i_B))$, and splicing them together can be done in constant time for each list, or $O(t_B)$ to splice all of them. Since $t_B \leq m_B$, splicing is less expensive than sorting.

Finding b_t, a_t

By doing a binary search, we can find b_t in an M_t list in $O(log(v))$ steps. Each step requires $O(p \cdot log(i))$ work to query the $\xrightarrow{hb}$ relationships with every message in B_{in}. Thus, the total work is $O(p \cdot log(v) log(i))$.

Competition Detection

With the sorted, spliced M_t lists, we can run the message detection algorithm from section 5.4.3. In the worst case, we must perform the same $O(fvc^3log(i))$ computation for each local execution. But we first try to reduce the lists by discarding external messages as described in section 7.3.3.

For a given function f with c inputs, suppose there are $c_B \leq c$ inputs for which messages of the appropriate types exist in BE. Starting at the tops of each of the M_t lists, we can check for $m_{tn} \xrightarrow{hb} min_{tx}$ for all min_{tx} messages in $O(c_Bp \cdot log(i))$. Then, traverse the lists from the bottoms and check for $max_{tx} \xrightarrow{hb} m_{tn}$, again in $O(c_Bp \cdot log(i))$.

If this effort is successful in removing a substantial portion of the messages in the M_t lists, then the total work will be $O(cvc_Bp \cdot log(i))$. Suppose that after removing these messages, the M_t lists contain only $n \ll cv$ messages. Then the work of the message competition algorithm is reduced to $O(fc^2n \cdot log(i))$. Unfortunately, $n \ll cv$ will not be true in general, so this work will not always help.

Chapter 8

Conclusion

8.1 Summary

In this dissertation, I have presented two models for distributed and concurrent systems that lend themselves to testing. The first is designed similar to message-passing environments such as MPI [44] and PVM [87]. It allows me to achieve coverage of a testing metric based on reversing pairs of messages in the system that can race with one another. A simplifying restriction is that processes in a correct program pass the same messages to the same receiving processes in the same order in any execution. This restriction allows expression of scatter-gather-style algorithms, but does not allow programming with an event-driven model. Two algorithms were presented:

- reversal of as many pairs of racing messages as possible in a single test run
- reversal of all racing message pairs in as few test runs as possible

In contrast with the model in chapter 3, the concurrent system model presented in chapter 4 is designed to support an event-driven programming style. The expectancy of determinism is re-assigned to the actual chain of causality expressed by the program text and the sender-receiver relationships created by message passing, rather than being fixed on individual processors or processes in the system. Also improved is the testing method. In this later research, I can prove conclusively that race conditions in the PUT will not cause behavior that I consider truly nondeterministic. Thus, the outcome for the given set of user-controllable inputs is

ensured to remain the same in any execution, and conventional testing techniques can be used to explore the effects of different user input valuations.

8.2 Future Research

The most obvious extension of this research is to build an implementation of the underlying runtime system and use it to solve some model problems. In doing so, a heuristic or an interface for receiving human input will be needed to replace the oracle used in boundary detection (section 7.1.3).

Another possibility is to relax the requirement of invocation atomicity. For systems that are designed to be deterministic, the atomicity of invocations is likely an unnecessary burden on the runtime system. In order for the system to be deterministic, invocations with variable conflicts cannot race, so there is no danger of interference. For systems with local nondeterminism it is harder to remove this requirement. Doing so would require exploring interleavings of multiple invocations' internal statements.

A more ambitious future goal would be to strategically select values for external inputs to the system (likely controllable, at least during testing) in an effort to find an execution that is not deterministic or converging. The methodology discussed in Chapter 4 builds knowledge of both the message-passing patterns and properties of the system's functions that might be useful in this effort.

Bibliography

[1] H. Agrawal, R. DeMillo, and E. Spafford. Dynamic slicing in the presence of pointers. In *ACM Symposium on Testing, Analysis, and Verification*, 1991.

[2] H. Agrawal, R. DeMillo, and E. Spafford. An execution backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.

[3] H. Agrawal, R. DeMillo, and E. Spafford. Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–619, June 1993.

[4] A. Aho, A. Dahbura, D. Lee, and M. Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. In S. Aggarwal and K. Sabnani, editors, *Proceedings of the IFIP WG 6.1, Eighth International Symposium on Protocol Specification, Testing, and Verification*, pages 75–86. Elsevier Science Publishers B.V. (North-Holland), June 1988.

[5] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, and causal. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 229–236, June 1995.

[6] G Avrunin, U Buy, T Corbett, L Dillon, and J Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, 1991.

[7] Adnan Aziz, Thomas R. Shiple, Vigyan Singhal, and Alberto L. Sangiovanni-Vincentelli. Formula-dependent equivalence for compositional ctl model checking. In *Lecture Notes in Computer Science, Proceedings of the 6th Conference on Computer-Aided Verification (CAV'94)*. Springer-Verlag, June 1994.

[8] A. Beguelin, J. Dongarra, G. A. Geist, R. Manchek, K. Moore, P. Newton, and V. Sunderam. Hence: A users' guide - version 2.0. Pvm Design and Visualization.

[9] Boris Beizer. *Black-Box Testing: Techiniques for Functional Testing of Software and Systems, 2nd ed.* John Wiley and Sons, Inc., 1990.

[10] J. R. Burch, E. M. Clarke, E. A. Emerson, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 1020 States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[11] D. Callahan and J. Subhlok. Static analysis for low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[12] Richard H. Carver and Kuo-Chung Tai. Use of sequencing constraints for specification. *IEEE Transactions on Software Engineering*, 24(6):471–490, June 1998.

[13] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, 1985.

[14] C. M. Chase and V. K. Garg. Efficient detection of restricted classes of global predicates. In Jean-Michel Hélary and Michel Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, volume 972, pages 303–317, Le Mont-Saint-Michel, France, 1995. Springer-Verlag.

[15] J-D Choi, B. P. Miller, and H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[16] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[17] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer-Verlag, 1982.

[18] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[19] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification*, pages 450–462. Springer-Verlag, 1993.

[20] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, May 1991.

[21] S. K. Damodaran-Kamal and J. M. Francioni. Nondeterminacy: Testing and debugging in message passing parallel programs. *SIGPLAN Notices: ACM/ONR Workshop on Parallel and Distributed Debugging*, 28(12):118–128, June 1993.

[22] S. K. Damodaran-Kamal and J. M. Francioni. Testing races in parallel programs with an OtOt strategy. In *International Symposium on Software Testing and Analysis*, pages 216–227, August 1994.

[23] Suresh K. Damodaran-Kamal and Joan M. Francioni. mdb: A semantic race detection tool for pvm. Technical report, University of Southwestern Louisiana, 1994.

[24] R. A. DeMillo. Testing Adequacy and Program Mutation. In *Proceedings of the 11th International Conference on Software Engineering*, pages 355–356, May 1989.

[25] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communication of the ACM*, 18(8):453–457, 1975.

[26] E. A. Emerson. Automated temporal reasoning about reactive systems. *Logics for Concurrency: Structure versus Automata. Lecture Notes in Computer Science: 1043*, pages 41–101, 1996.

[27] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification*, pages 463–478. Springer-Verlag, 1993.

[28] J. Esparza. Model checking using net unfoldings. In *Science of Computer Programming*, volume 23, pages 151–195, 1994.

[29] S. Feldman and C. Brown. IGOR: a system for program debugging via reversible execution. *SIGPLAN Notices: Workshop on Parallel and Distributed Debugging*, 24(1):112–123, May 1988.

[30] Jean-Claude Fernandez, Claude Jard, Thierry Jeron, and Cesar Viho. Using on-the-fly verification techniques for the generation of test suites. In *Computer Aided Verification, 8th International Conference*. Springer, July 1996.

[31] M. G. Fernandez and S. Ghosh. Ddbx-lpp: A dynamic software tool for debugging asynchronous distributed algorithms on loosely coupled parallel processors. *Journal of Systems and Software*, 22(1):27–43, July 1993.

[32] C. J. Fidge. Partial orders for parallel debugging. *SIGPLAN Notices: Workshop on Parallel and Distributed Debugging*, 24(1):183–194, January 1989.

[33] G. C. Fox. *Solving problems on concurrent processors*. Prentice Hall, 1988.

[34] V. K. Garg, C. Chase, J. R. Mitchell, and R. Kilgore. Detecting conjunctive channel predicates in a distributed programming environment. In *28th Hawaii International Conference on System Sciences*, pages 232–241, January 1995.

[35] V. K. Garg and C. M. Chase. Distributed algorithms for detecting conjunctive predicates. Technical Report ECE-PDS-94-03, University of Texas at Austin, ECE Dept., 1994.

[36] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, 1994.

[37] O. Gerstel, M. Hurfin, N. Plouzeau, M. Raynal, and S. Zaks. On-the-fly replay: A practical paradigm and its implementation for distributed debugging. In *Proceedings, Symposium on Parallel and Distributed Processing*, pages 266–272, October 1994.

[38] Carlo Ghezzi, Dino Mandrioli, Sandro Morasca, and Mauro Pezze. Symbolic execution of concurrent systems using petri nets. *Computer Languages*, 14(4):263–281, 1989.

[39] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings, 2nd Workshop on Computer Aided Verification, Lecture notes in Computer Science*, volume 531, pages 176–185. Springer-Verlag, June 1990.

[40] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An approach to the State-Explosion Problem*, volume 1032. Springer-Verlag, January 1996.

[41] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, July 1991.

[42] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings, 3rd Workshop on Computer Aided Verification, Lecture notes in Computer Science*, volume 575, pages 332–342. Springer-Verlag, July 1991.

[43] S. Grabner and J. Volkert. Debugging parallel programs using event graph manipulation. In *International Meeting on Vector and Parallel Processing*, pages 443–450, October 1993.

[44] gropp@@mcs.anl.gov and lusk@@mcs.anl.gov. MPI : The Message Passing Interface. http://www-unix.mcs.anl.gov/mpi/.

[45] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[46] G. J. Holtzmann and D. Peled. An improvement in formal verification. In *Proceedings, FORTE '94*, pages 177–191, 1994.

[47] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Proceedings of Supercomputing*, August 1990.

[48] William E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, New York, 1987.

[49] C. N. Ip and D. L. Dill. Better verification through symmetry. In *Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications*, pages 97–111, 1993.

[50] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Journal of Algorithms*, 1990.

[51] S. Kashyap and V. K. Garg. Exploiting predicate structure for efficient reachability detection. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, November 2005.

[52] S. Kashyap and V. K. Garg. Intractability results in predicate detection. *Information Processing Letters*, 94(6):277–282, June 2005.

[53] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 489–507. Springer-Verlag, 1989.

[54] R. Kilgore and C. Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. In *30th Hawaii International Conference on System Sciences*, 1997.

[55] Richard B. Kilgore and Craig M. Chase. Testing distributed programs containing racing messages. *The Computer Journal*, 40(8):489–498, February 1997.

[56] J. Kohl. XPvm. http://www.netlib.org/utk/icl/xpvm/ xpvm.html.

[57] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[58] T. J. LeBlanc and J. M. Mellor-Crummley. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[59] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, January 1985.

[60] G. Luo, G. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, 1994.

[61] Z. Mana and A. Pnueli. Temporal verification diagrams. *Theoretical Aspects of Computer Software. Lecture Notes in Computer Science: 789*, pages 726–765, 1994.

[62] Z Manna and Amir Pnueli. Verification of concurrent programs, the temporal framework. Technical report, Stanford University, Stanford, CA, USA, 1981.

[63] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[64] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course, Lecture Notes in Computer Science*, volume 255, pages 279–324. Springer-Verlag, 1986.

[65] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings, 4th Workshop on Computer Aided Verification, Lecture Notes in Computer Science*, volume 663, pages 164–177. Springer-Verlag, June 1992.

[66] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, 1993.

[67] N. Mittal and V. K. Garg. Debugging distributed programs using controlled reexecution. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 239–248, New York, NY, 2000. ACM Press.

[68] N. Mittal and V. K. Garg. Computation slicing: Techniques and theory. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 78–92, London, UK, 2001. Springer-Verlag.

[69] Sandro Morasca and Mauro Pezze. Using high-level petri nets for testing concurrent and real-time systems. In H. Zedan, editor, *Proceedings of Real-Time Systems: Theory and Appplications*, pages 119–131. British Computer Society, York, September 1989.

[70] Tadao Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[71] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[72] R. Netzer, T. Brennan, and S. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Symposium on Parallel and Distributed Tools*, pages 31–40, May 1996.

[73] R. Netzer and B. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, August 1990.

[74] R. Netzer and B. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings, Supercomputing '92*, pages 502–511, November 1992.

[75] R. Netzer and B. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *Journal of Supercomputing*, 8(4):371–388, 1995.

[76] Robert H. B. Netzer, Sairam Subramanian, and Jian Xu. Critical-path-based message logging for incremental replay of message-passing programs. Technical report, Brown University, 1994.

[77] Robert H. B. Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. Technical report, Brown University, 1993.

[78] S. C. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988.

[79] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. Procedures for reducing the size of coverage-based test sets. Technical Report ECE-PDS-94-03, George Mason University, PRC Inc., Reliable Software Technologies Corp., 1994.

[80] Thomas J. Ostrand and Marc J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[81] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.

[82] D. Z. Pan and M. A. Linton. Supporting reverse execution of parallel programs. *SIGPLAN Notices: 1988 Workshop on Parallel and Distributed Debugging*, 24(1):124–129, January 1989.

[83] D. Peled. All from one, one for all: on model checking using representatives. In *Proceedings, 5th Conference on Computer Aided Verification, Lecture notes in Computer Science*, volume 697, pages 409–423. Springer-Verlag, June 1993.

[84] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings, 6th Conference on Computer Aided Verification, Lecture notes in Computer Science*, volume 818, pages 377–390. Springer-Verlag, June 1994.

[85] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8:39–64, 1996.

[86] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[87] pvm@@msr.epm.ornl.gov. PVM : Parallel Virtual Machine. http://www.epm.ornl.gov/pvm/pvm_home.html.

[88] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351. Springer-Verlag, 1982.

[89] D. A. Reed. An overview of the pablo performance analysis environment. In *Technical Report, University of Illinois, Urbana, Illinois*, November 1992.

[90] W. Reisig. *Petri Nets: An Introduction.*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[91] D. Russell. State restoration in systems of communicating processes. In *IEEE Transactions on Software Engineering*, March 1980.

[92] K Sabnani and A Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15(4):285–297, 1988.

[93] R. S. Side and G. C. Shoja. A debugger for distributed programs. *Software - Practice and Experience*, 24(5):507–525, May 1994.

[94] A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. In *Proceedings of the IEEE 9th Symposium on Parallel and Distributed Processing (SPDP)*, pages 763–769, Orlando, 1998.

[95] Ashis Tarafdar and Vijay K. Garg. Happened before is the wrong model for potential causality. Technical Report TR-PDS-98-006, The University of Texas at Austin, 1998.

[96] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.

[97] A. Valmari. Error detection by reduced reachability graph generation. In *Proceedings, 9th International Conference on Application and Theory of Petri Nets*, pages 95–112, 1988.

[98] A. Valmari. Heuristics for lazy state generation speeds up analysis of concurrent systems. In *Proceedings of the Finish Artificial Intelligence Symposium STeP-88*, volume 2, pages 640–650, 1988.

[99] A. Valmari. A stubborn attack on state explosion. In *Lecture Notes in Computer Science*, volume 531, pages 156–165. Springer-Verlag, June 1990.

[100] A. Valmari. On-the-fly verification with stubborn sets. In *Proceedings, 5th Conference on Computer Aided Verification*, volume 697, pages 397–408, June 1993.

[101] Antti Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd Workshop on Computer-Aided Verification, vol 531 of Lecture Notes in Computer Science*, pages 156–165. Springer-Verlag, June 1990.

[102] S. N. Weiss and Elaine J. Weyuker. An Extended Domain-based Model of Software Reliability. *IEEE Transactions on Software Engineering*, 14(10):1512–1524, October 1988.

[103] Elaine J. Weyuker. The Evaluation of Program-based Software Test Data Adequacy Criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[104] G. Winskel. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course, Lecture Notes in Computer Science*, volume 255, pages 325–392. Springer-Verlag, 1986.

Vita

Richard Brian Kilgore was born in Fort Worth, Texas on January 29, 1968, the son of Geraldine Francis Kilgore and Webster Calvin Kilgore. After graduating from Annandale High School, Annandale, Virginia, in 1986, he entered The University of Virginia in Charlottesville, Virginia. He received a Bachelor of Science from The University of Virginia in May 1990. After 3 years in working in industry for the MITRE Corporation in McLean, Virginia, Richard entered the Graduate School of The University of Texas in 1993, and earned a Master of Science in Computer engineering in May 1996.

While earning a Doctor of Philosophy from The University of Texas at Austin, Richard has worked as a Software Engineer, and now works for Amazon.com in Seattle, Washington.

Permanent Address: 4508 183rd Pl SW
Lynnwood, WA 98037

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.