

Practical Concurrency Testing

Ao Li

CMU-S3D-26-108

May 2026

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Rohan Padhye (Co-Chair)
Vyas Sekar (Co-Chair)
Ben Titzer
Shan Lu (Microsoft Research)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.*

Copyright © 2026 **Ao Li**

This work was partially conducted at Amazon as part of an AWS Applied Science internship. This work was supported in part by an Amazon Research Award, in part by the National Science Foundation under grants CCF-2120955, CCF-2429384, and CCF-2453432, and in part by DSO National Laboratories. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, organization, or government agency.

Keywords: Concurrency, Testing, Debugging

To all sudoers.

Abstract

Cloud services have become critical infrastructure, yet long-studied concurrency bugs continue to cause real-world failures. Ideally, we want a system for concurrency testing that offers reliable control over schedules, is efficient, and can handle real-world programs at scale. Unfortunately, existing approaches usually have to sacrifice one or more of these.

The key insight in this thesis is to adopt a behavior-centric view of concurrency testing. Rather than faithfully reproducing the exact execution environment of a concurrent or distributed program, I argue that what matters for testing is preserving and exploring the space of observable program behaviors. Guided by this principle, I design and implement three systems, Fray, Fest, and Diorama. Fray serializes multithreaded JVM programs into controlled sequential executions using shadow locks, uncovering 19 real bugs in systems like Kafka and Lucene. Fest applies greybox fuzzing-inspired feedback to prioritize schedules likely to reveal new behaviors, achieving up to 41x higher behavioral coverage. Diorama extends this approach to distributed systems by replacing the network layer with in-memory message queues and wall-clock time with a logical clock, reducing distributed executions to controllable single-process ones.

Acknowledgments

If you are reading this, then this acknowledgment is for you. Thank you.

Contents

1	Introduction	1
1.1	Scope	2
1.2	Objective	2
1.2.1	Deterministic and General-purpose Schedule Execution	2
1.2.2	Efficient Exploration of Schedules	3
1.2.3	Reducing Distributed Systems to Concurrent Applications	4
1.3	Thesis Statement	4
1.4	Contributions	4
2	Background and Related Work	6
2.1	Concurrent Programs	6
2.2	Formal Semantics	6
2.2.1	Scheduler	9
2.2.2	Concurrency Testing Algorithm	10
2.3	Evolution in Concurrency Testing Platform	10
2.4	Evolution in Testing Algorithms	11
2.5	Distributed System Testing	12
3	FRAY: An Efficient General-Purpose Concurrency Testing Platform for the JVM	14
3.1	Design of FRAY	16
3.1.1	Concurrency Control in FRAY	17
3.1.2	Soundness and Completeness	20
3.1.3	Scheduling and Search Strategies	21
3.2	Apply FRAY to Real-World Software	22
4	FEST: Feedback-guided Adaptive Testing of Distributed Systems Designs	24
4.1	Background	26
4.1.1	P Framework	26
4.1.2	Schedules and Randomized Search-based Bounded Concurrency Testing	28
4.1.3	Deterministic Record and Replay	29
4.1.4	Behavioral Coverage for Distributed Systems	30
4.2	Problem and Solution Overview	30
4.2.1	Problem Statement	30
4.2.2	Feedback-Guided Adaptive Scheduling	30

4.2.3	Scenario-Guided Schedule Generation	32
4.2.4	Scope and Limitations	32
4.3	FEST Design	33
4.3.1	Main Algorithm	33
4.3.2	Adaptive Schedule Generation	34
4.3.3	System Execution	36
4.3.4	Diversity Feedback	37
4.3.5	Scenario Specification Language	38
4.3.6	Compliance Feedback	39
4.4	Evaluation	39
4.4.1	Implementation	39
4.4.2	Comparison and Benchmarks	39
4.4.3	RQ1: Timeline Coverage Improvement	40
4.4.4	RQ2: Bug Finding Capabilities	42
4.4.5	RQ3: Scenario Coverage	43
4.4.6	RQ4: Priority-based Schedule Generation	44
5	Diorama: Microcosm Distributed System Testing	46
5.1	Background	47
5.1.1	The AWS Route53 Incident	47
5.1.2	Timed Operations	48
5.2	Design	50
5.2.1	Network Virtualization	51
5.2.2	Time Virtualization	52
5.2.3	Node Isolation via Class Loading	53
5.2.4	Lifecycle	53
5.3	Case Study	54
5.3.1	Implementation	54
5.3.2	Test Setup	55
5.3.3	Results	55
5.4	Use DIORAMA to Analyze Real-World Distributed Systems	57
5.4.1	Adapting Real-World Systems to DIORAMA	57
5.4.2	Manual Test Development: Apache ZooKeeper	57
5.4.3	LLM-Assisted Test Generation	59
5.4.4	LLM-Generated Test Suites	60
5.4.5	Limitations and Future Work	61
6	Conclusion and Discussion	63
6.1	Summary	63
6.2	Discussion: Where Are We Heading?	63
	Bibliography	65

List of Figures

2.1	Sample Java program containing several concurrency primitives. The program is well synchronized (i.e., no data races) but has concurrency bugs: it can non-deterministically run to completion, deadlock, or trigger an assertion violation. .	7
3.1	FRAY introduces a new abstraction level for concurrency control called <i>shadow locking</i> , which identifies a sweet spot in the trade-off between performance overhead, the size of the scheduling search space, and general-purpose applicability.	16
3.2	FRAY’s instrumentation (highlighted) of the program depicted in Fig. 2.1, demonstrating <i>shadow locking</i> for concurrency control. Each $\mathcal{S}_*^t$ is a lock instantiated dynamically by FRAY and originally held by FRAY’s scheduler thread; the shadow lock will be released by FRAY when it wants to schedule thread t	18
4.1	Overview of the search performed by RSBT and feedback-guided adaptive RSBT. Each edge connects two schedules, indicating that the target schedule is generated adaptively based on the source schedule.	25
4.2	A distributed storage system with four replicas.	26
4.3	Replica state machine in P.	27
4.4	Scenario specification language and example scenarios for the key-value store in § 4.2.3	38
4.5	Cumulative distribution function of improvements over the original approach without feedback (e.g., FES _T PCT ₃ versus PCT ₃) for different schedule generation algorithms.	41
4.6	Best performance count: Number of test cases for which each technique achieves the highest timeline coverage.	42
4.7	Upset plot for bug counts across different scheduling algorithms. Higher is better.	43
4.8	Comparison of timeline coverage that satisfies the specified scenario across different techniques.	43
4.9	Timeline coverage improvement of FES _T PCT ₁₅ compared to the version without priority-based scheduling (NoPrio). The y-axis is cut off at 2× due to space limitations.	45

5.1	An illustration of the AWS Route53 race condition caused by a complex interaction between network messages and thread interleavings. Two Enactor nodes send different configuration versions (v1 and v2) to the Route 53 node. Worker threads within Route 53 process these updates concurrently, leading to a race condition where operations on different versions interleave incorrectly. The red bug icon marks the point where the inconsistency is detected during a version update. . . .	48
5.2	Architecture of DIORAMA. Each logical node runs in an isolated class-loader within a single JVM. The network layer is replaced by in-memory message queues, and wall-clock time is replaced by a logical clock. A unified scheduler controls thread interleavings, message delivery, and time advancement.	51
5.3	Per-machine time thread. Each logical node maintains its own logical clock, incremented by a dedicated background thread whose scheduling is controlled by DIORAMA.	52
5.4	Virtualized sleep. DIORAMA replaces <code>Thread.sleep()</code> with a yield loop that waits for the logical clock to reach the target deadline.	52
5.5	Categorical heatmap of happens-before pairs explored by each tool. Orange cells indicate pairs explored only by DIORAMA; dark grey cells indicate pairs explored by both tools; light grey cells indicate pairs explored by neither tool. No pairs are explored exclusively by <code>rr</code> (blue). DIORAMA explores a strict superset of the orderings that <code>rr</code> reaches.	56

List of Tables

- 2.1 Comparison of representative distributed-system testing techniques. The key distinction is which sources of nondeterminism each approach can directly control while remaining efficient enough for systematic exploration. 12
- 3.1 Results of concurrency testing on real-world software. “Tests run” represents the number of test cases that can be executed and whose thread interleavings searched over without internal tool errors. The second column shows the number of test failures identified by each technique (and the subset of failures that are caused by wall-clock-based timed waits). The last column presents the total number of reported bugs (along with the subset confirmed by the developers). 22
- 4.1 Description of case studies from AWS and open-source models. ‘LOC’ indicates the P line of code of the formal model of the system. 40
- 4.2 Description of testing scenarios. 41
- 5.1 Happens-before pair exploration across 1,000 iterations. DIORAMA explores a strict superset of the orderings reached by `rr`, covering 47 additional pairs. . . . 55
- 5.2 Target systems and the concurrency properties tested by the LLM-generated harnesses. 60

Chapter 1

Introduction

Software testing is the predominant form of validating correctness for large real-world programs due to its simplicity, widespread applicability, efficiency, and reproducibility. However, testing *multi-threaded* programs remains challenging in practice, despite the fact that concurrency bugs are among the most difficult to detect and diagnose [46, 54].

Take Java, which by several metrics is the most popular programming language with native support for concurrent multi-threading [27, 66, 69]. Let’s assume that programmers write test cases to validate the correctness of concurrent programs via simple assertions (e.g., that a *parallel-sort* operation produces a sorted list). How can we check such properties? In practice, a common approach appears to be simply re-running concurrent tests multiple times [10] to check whether some assertion fails. This approach is neither efficient nor reproducible. For example, in Lucene, the most popular open-source search engine maintained by Apache Foundation, thousands of stress tests are still used to test concurrent components. Even more problematic, many of these tests are flaky due to the inherent nature of concurrency, leading to unresolved test failures that persist for years [48, 49].

The central insight of this thesis is that effective concurrency testing does not require faithfully reproducing the exact execution environment of the target program. What matters is preserving and exploring the space of *observable program behaviors*. Faithful execution—running threads on real OS schedulers, sending messages over real networks, waiting for real wall-clock time—is neither necessary nor sufficient for effective testing: it is expensive to control and explores only a narrow slice of the behavioral space. By adopting this behavior-centric view, we can build testing systems that are simultaneously more controllable, more efficient, and more effective than approaches that attempt to test programs in their native execution environments.

Guided by this principle, we make three contributions. First, we show that a concurrent program does not need to run concurrently to exhibit all of its concurrent behaviors: by adding shadow locks that serialize thread execution, we can systematically explore every observable interleaving (FRAY, Chapter 3). Second, we show that observed behaviors should guide which schedules to explore next: by using behavioral feedback from past executions, we can prioritize schedules that are more likely to uncover new behaviors (FEST, Chapter 4). Third, we show that a distributed system does not need to run as a true distributed deployment to be tested effectively: by reducing network communication and time to shared-memory abstractions, we can test distributed systems as single-process concurrent programs while preserving all application-

observable behaviors (DIORAMA, Chapter 5).

1.1 Scope

Our goal is to find *concurrency bugs* in concurrent and distributed software written in Java. We begin with multi-threaded Java programs, where concurrency bugs arise from thread interleavings within a single process, and then extend our scope to distributed systems written in Java, where bugs can additionally arise from the interplay of message ordering and timing across networked nodes. In general, a concurrency bug is usually caused by a *race condition*—that is, only manifests under certain interleavings of threads or message deliveries. Following the style in the seminal work by Lu et al. [46], we use *concurrency bug* to include atomicity violations, ordering violations, deadlocks, etc. but not *data races*. This nuance is subtle but important.

In Java, the term *data race* refers to concurrent conflicting accesses (i.e., one write and another read/write) to a non-volatile shared variable. Java’s weak memory model allows programs with data races to exhibit behavior that cannot be explained by *any* sequence of thread interleavings [29]. Data races can be effectively identified by race detectors [24, 26, 55, 64, 65] and can be easily fixed by using proper synchronization to control access to shared memory [45]. For our purposes, we assume that programs are *free of data races* since developers can use the aforementioned techniques to remove them.

Concurrency bugs such as atomicity and ordering violations occur commonly in well-synchronized programs [47] and are much harder to debug since they cannot be addressed simply by adding more locks. Data-race-free Java programs exhibit sequential consistency [29]; so, we can explain concurrency bugs to developers by demonstrating a specific sequence of thread interleavings called a *schedule*.

1.2 Objective

1.2.1 Deterministic and General-purpose Schedule Execution

A concurrent program does not need to run concurrently to exhibit concurrent behaviors. The observable behaviors of a multithreaded program—the assertion outcomes, outputs, and exceptions it can produce—are determined by which thread interleavings are explored, not by whether threads execute in parallel. This means we can serialize a concurrent program into a sequential execution, choosing at each scheduling point which thread runs next, and still observe every behavior that the original concurrent execution could produce.

General-Purpose Push-Button End-to-End (E2E) Testing We want a concurrency testing system that can scale to real-world software and work on off-the-shelf JVM programs. This means (a) targeting general-purpose concurrency bugs expressed as arbitrary program assertions, instead of only checking specific properties such as linearizability [31, 37] or class thread-safety [44]; (b) eliminating the need for developers to manually set up concurrency testing, such as writing specifications in a DSL, rewriting source code to use mocked concurrency libraries, or

implementing support for specific frameworks—ideally, we want a drop-in wrapper for the `java` command; and (c) being able to run on mature real-world software—the latter requires designing a system such that the engineering effort (on the tool maintainer’s part) to support a variety of application uses is minimized in practice.

Application-level Concurrency Testing In order to achieve efficient concurrency testing for JVM programs, we need to achieve two goals. First, we want to minimize performance overhead when deterministically executing a program along a fixed thread schedule. Typically, this means executing at least the non-synchronizing program instructions (e.g., memory reads/writes) at the same speed as during normal execution. Note that executing a multi-threaded program by scheduling threads one at a time sequentially is acceptable; during concurrency testing, we can parallelize the search algorithms by running different schedules across available CPUs, which maintains overall testing throughput (i.e., execs/time).

Second, for testing Java-like programs that run in a managed runtime (i.e., the JVM), we want to optimize the search over thread schedules to only consider interleavings within application threads. This means abstracting away the non-determinism within the managed runtime (e.g., VM initialization, garbage collection, class loading).

Deterministic and Reproducible Our system must ensure deterministic and reproducible concurrency testing. Specifically, given a schedule, the testing framework should faithfully execute it in a deterministic manner. Furthermore, repeated executions of the same schedule should consistently produce identical results, ensuring reliable debugging and verification.

In Chapter 3, we discuss the design and implementation of FRAY [42], an efficient general-purpose concurrency testing platform for the JVM.

1.2.2 Efficient Exploration of Schedules

Systematic testing and model checking are theoretically ideal for verifying the correctness of concurrent programs, as they aim to exhaustively explore all possible schedules [40]. However, for real-world concurrent programs, these approaches are impractical due to the immense size of the schedule space, which makes it infeasible to evaluate every possible interleaving of threads. Consequently, efficient schedule exploration becomes a necessity for building a practical concurrency testing platform, especially when operating within constrained time and computational budgets.

The behavior-centric view suggests a natural solution: use observed behaviors to guide which schedules to explore next. Rather than sampling schedules blindly or relying solely on structural constraints, we can evaluate each executed schedule by the behaviors it produces and prioritize future exploration toward schedules that are likely to exhibit new, previously unseen behaviors. When the system discovers schedules that produce diverse behavioral patterns, it mutates and extends them; when a region of the schedule space yields only redundant behaviors, it moves on. This feedback loop—analogueous to coverage-guided fuzzing in sequential testing—transforms schedule exploration from a blind search into an adaptive, behavior-directed process.

In Chapter 4, we discuss the design and implementation of FEST [43], an efficient adaptive scheduler that aims to maximize behavioral coverage and bug discovery for concurrent programs.

1.2.3 Reducing Distributed Systems to Concurrent Applications

Having established a reliable foundation for controlling and testing concurrent programs, a natural question arises: can this testing power be extended to distributed systems? Unlike single-process applications, distributed systems are subject to three tightly coupled sources of nondeterminism: thread interleavings within each node, message interleavings across nodes, and time-dependent behavior such as delays, timeouts, and timer-triggered events.

The behavior-centric view offers a powerful answer: testing a distributed system does not require faithfully executing it as a true distributed deployment. From the application’s perspective, network communication and time are observable only through their effects on application state—messages arrive and modify local data structures, clocks advance and trigger timeouts. These effects can be modeled entirely through shared-memory abstractions: message passing becomes reads and writes to in-memory queues, and wall-clock time becomes accesses to a shared monotonic counter. This lets us reduce a distributed execution to a single-process, multi-threaded execution that preserves all application-observable behaviors while making the problem dramatically easier to control and explore. As a result, the rich schedule exploration capabilities developed for single-process programs—including FRAY’s deterministic execution and FEST’s adaptive scheduling—apply directly to distributed systems.

In Chapter 5, we present DIORAMA, a *microcosm framework* that realizes this reduction. By replacing the network layer with in-memory message queues and real time with a shared counter, DIORAMA turns message delivery and time advancement into scheduler-controlled choices, enabling systematic joint exploration of thread, message, and timing nondeterminism.

1.3 Thesis Statement

By focusing on observable program behaviors rather than faithful execution, we can build concurrency and distributed system testing tools that are simultaneously more controllable, more efficient, and more effective than existing approaches.

1.4 Contributions

The three contributions of this thesis form a natural progression from local concurrency to models of distributed systems and finally to real-world distributed system testing, with each piece enabling and complementing the next. FRAY establishes the foundation: a mechanism for deterministically controlling and exploring thread interleavings within a single process. FEST builds on this foundation by making schedule exploration intelligent—using behavioral feedback from past executions to efficiently navigate the vast space of possible interleavings without exhaustive enumeration. DIORAMA then extends the reach of both techniques to distributed systems by reducing network communication and time to shared-memory abstractions, turning a distributed

deployment into a single-process concurrent program that can be tested using the same controlled execution and adaptive exploration developed in the first two contributions. Together, these three pieces demonstrate that the behavior-centric principle applies uniformly across the spectrum—from local thread interleavings to distributed system behaviors—and that advances at one level of the progression directly enable advances at the next.

Chapter 2

Background and Related Work

2.1 Concurrent Programs

A concurrent program consists of multiple threads or processes that execute independently but may interact or share resources, such as memory, files, or network connections. These interactions and shared resource accesses require coordination, often through synchronization mechanisms, to ensure correctness. Concurrent programs are common in modern software systems, where tasks like handling multiple user requests, processing data streams, or performing parallel computations are executed simultaneously for improved performance and responsiveness.

Concurrent programming introduces non-determinism because the execution order of operations across threads or processes cannot be guaranteed and depends on *scheduling* by the underlying system. This non-determinism makes concurrent programs inherently more complex and challenging to design, debug, and test than sequential programs.

Figure 2.1 depicts a sample (data-race-free) Java program which spawns two threads (Line 18), one of which sets local `x=0` and the other sets local `x=1` (Line 6). The program often terminates successfully; however, it can non-deterministically deadlock or trigger an assertion violation. For example, if the thread that calls `notify()` (Line 11) enters the `synchronized` block (Line 7) before the thread that calls `wait()` (Line 9), then the second thread will wait forever in a deadlock. Otherwise, if the thread that calls `notify()` enters the `synchronized` block second and then updates the shared `volatile` variable `b` (Line 14) before the thread waking from `wait()` can do so, then the value of `b` will end up being 0, triggering an assertion failure (Line 20).

2.2 Formal Semantics

Definition 1 (Formal Model of Java Programs). *A Java program can be modeled as a tuple*

$$P = \langle Ins, Tid, PC, \mathcal{M}, Lid, Cid, exec, update \rangle,$$

- $Ins = Local \cup Atomic \cup \{threadStart\} \times PC$
 $\cup \{new\ Lock, lock, unlock\} \times Lid \cup \{new\ Cond _ from _ \} \times Cid \times Lid$
 $\cup \{await, signal, signalAll\} \times Cid \cup \{undef\}$ is a set of instructions. The sets

```

1  class Foo extends Thread {
2      static Object o = new Object();
3      static AtomicInteger a = AtomicInteger();
4      static volatile int b;
5      public void run() {
6          int x = a.getAndIncrement();
7          synchronized(o) {
8              if (x == 0) {
9                  o.wait();
10             } else {
11                 o.notify();
12             }
13         }
14         b = x;
15     }
16     public static void main(...) {
17         Foo[] threads = {new Foo(), new Foo()};
18         for (var thread : threads) thread.start();
19         for (var thread : threads) thread.join();
20         assert (b == 1);
21     }
22 }

```

Figure 2.1: Sample Java program containing several concurrency primitives. The program is well synchronized (i.e., no data races) but has concurrency bugs: it can non-deterministically run to completion, deadlock, or trigger an assertion violation.

Local and Atomic represent all local and atomic instructions, respectively. A local instruction may read or write local variables, while an atomic instruction may access atomic variables in addition to local variables. For simplicity, we do not model each of these instructions and instead assume abstract properties for them. Assuming sequential consistency, volatile memory access is treated as an atomic instruction. We plan to refine this model to include additional synchronization primitives: `wait`, `notify`, `notifyAll`, `LockSupport.park()`/`unpark()`, and `Thread.interrupt()`.

- *Tid is a set of thread identifiers with a special value $\text{main} \in \text{Tid}$ denoting the identifier for the main thread.*
- *PC is a set of program counters with special value $pc_{\text{none}}, pc_{\text{init}} \in \text{PC}$, which denote thread termination and the initial program counter, respectively.*
- *$\mathcal{M}$ is a set of memory states with a special value $M_{\text{init}} \in \mathcal{M}$ denoting the initial memory configuration.*
- *Lid and Cid are sets of lock identifiers and condition identifiers, respectively.*
- *$\text{exec} : \text{PC} \rightarrow \text{Ins} \times \text{PC}$ is an execution function which takes a program counter and returns the instruction and the next program counter.*
- *$\text{update} : \text{Ins} \times \mathcal{M} \rightarrow \mathcal{M}$ represents the effect of an instruction on a memory state.*

Definition 2 (Formal Model of Transition System Induced by a Java Program). *Given a Java*

program P , we define a transition system induced by P as follows:

$$\mathcal{T}_P = \langle S, \text{Act}, \longrightarrow, s_{\text{init}} \rangle.$$

- S is a set of states consisting of tuples of the form $\langle T, M, L, C \rangle$.
 - $T \in \text{Tid} \rightarrow_{\text{fin}} PC$ is a finite mapping from thread identifiers to program counters
 - $M \in \mathcal{M}$ is a memory state
 - $L \in \text{Lid} \rightarrow_{\text{fin}} \text{Tid} \times \mathbb{N}_{\geq 0}$ is a finite mapping from lock identifiers to lock states. Each lock state is a tuple $\langle t, n \rangle$ where $t \in \text{Tid} \cup \{\text{none}\}$ is the thread that holds the lock and n is the hold counter.¹
 - $C \in \text{Cid} \rightarrow_{\text{fin}} \text{Lid} \times 2^{\text{Tid}}$ is a finite mapping from condition identifiers to condition states. Each condition state is a tuple $\langle r, S, H \rangle$, where $r \in \text{Lid}$ is the lock associated with the condition, $S \subseteq \text{Tid}$ is a set of threads that are waiting on the condition, and $H \in \text{Tid} \rightarrow_{\text{fin}} \mathbb{N}_{\geq 0}$ is a finite partial function recording the hold counter of the lock for each thread yielded by calling `await`.
- Act is a set of actions of the form $\langle t, e \rangle$ where $t \in \text{Tid}$ is a thread identifier and $e \in \{\text{local}, \text{atomic}, \text{thread_start}, \text{lock_create}, \text{lock}, \text{unlock}, \text{cond_create}, \text{await}, \text{awake}, \text{awake_spurious}, \text{signal_all}\} \cup \{\text{signal}\} \times \text{Tid}$ is an effect representing the type of transition.
- $\longrightarrow \subseteq S \times \text{Act} \times S$ is a transition relation defined below
- $s_{\text{init}} = \langle T_0, M_0, L_0, C_0 \rangle \in S$ is the initial state with $T_0 = \{\text{main} \mapsto pc_{\text{init}}\}$, $M_0 = M_{\text{init}}$, $L_0 = C_0 = \emptyset$.

Throughout this thesis, we assume that a Java program P is given, and we use $\mathcal{T}$ to denote the transition system induced by it.

Definition 3 (Thread function). We use a function $\text{thread} : \text{Act} \rightarrow \text{Tid}$ to denote the thread identifier associated with each action, that is, $\text{thread}(\langle t, e \rangle) = t$.

A schedule is a sequence of states in the transition system. This is the main building block of our work.

Definition 4 (Schedule). A schedule of the transition system is a sequence of states $s_0, s_1, \dots, s_n$ and actions $\alpha_0, \alpha_1, \dots, \alpha_{n-1}$ satisfying $s_i \xrightarrow{\alpha_i} s_{i+1}$ for each $i = 0, 1, \dots, n-1$. We denote a schedule by

$$\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{n-1}} s_n.$$

If the actions are not used in the proof, we omit them and write:

$$\pi = s_0 \longrightarrow s_1 \longrightarrow \dots \longrightarrow s_n$$

for simplicity. We treat s_0 as a schedule of length 0.

The following are key lemmas required to justify definitions (Definition 7 and 8) below.

Lemma 5 (Transition is Action Deterministic). Given a state s and an action α , there exists at most one s' such that $s \xrightarrow{\alpha} s'$.

Proof. Let $s = \langle T, M, L, C \rangle$. Suppose to the contrary that there exist two states s_1 and s_2 such that $s \xrightarrow{\langle t, e \rangle} s_1$ and $s \xrightarrow{\langle t, e \rangle} s_2$.

¹We need a hold counter because Java locks are reentrant.

First, suppose that the transitions $s \xrightarrow{\langle t, e \rangle} s_1$ and $s \xrightarrow{\langle t, e \rangle} s_2$ are derived by the same rule. Then, we can easily check that $s_1 = s_2$ by construction.

We must consider the remaining cases, specifically the pairs of rules that derive transitions with the same action.

lock₁ and lock₂ Suppose that $s \xrightarrow{\langle t, e_1 \rangle} s_1$ and $s \xrightarrow{\langle t, e_2 \rangle} s_2$ are derived by the rule *lock₁* and *lock₂*, respectively. Note that r used in the rules are uniquely determined by *ins* and t is also unique. Then $L(r) = \langle \text{none}, 0 \rangle$ and $L(r) = \langle t, 1 \rangle$ by the premise of the rules, respectively. This is a contradiction.

unlock₁ and unlock₂ Suppose that $s \xrightarrow{\langle t, e_1 \rangle} s_1$ and $s \xrightarrow{\langle t, e_2 \rangle} s_2$ are derived by the rule *unlock₁* and *unlock₂*, respectively. Note that r used in the rules are uniquely determined by *ins* and t is also unique. By the premise of the rules, we have $L(r) = \langle t, 1 \rangle$ and $L(r) = \langle t, n \rangle$ for some $n > 1$, which is a contradiction.

□

Using Lemma 5, we can define useful notations as follows:

Definition 6 (Enabled Actions). *For a state s , the set of enabled actions $enabled(s)$ is defined as follows:*

$$enabled(s) = \{\alpha \in Act \mid \exists s'. s \xrightarrow{\alpha} s'\}$$

Definition 7 (Next Actions for a Thread). *We use $next(s, t)$ to denote the set of actions (t, e) enabled in s . That is, $next(s, t) = \{\alpha \in enabled(s) \mid \alpha = (t, e) \text{ for some } e\}$.*

Definition 8 (Application of an Action to a State). *By Lemma 5, for a state s and $\alpha \in enabled(s)$, there exists a unique state s' such that $s \xrightarrow{\alpha} s'$. We use $\alpha(s)$ to denote the state s' . If such a state does not exist, i.e., if $\alpha \notin enabled(s)$, we write $\alpha(s) = \text{undefined}$.*

Definition 9 (assertion). *An assertion a is a function $a : \mathcal{M} \rightarrow \mathbb{B}$ that evaluates a given memory state and returns a boolean value indicating whether the condition it represents holds in that state.*

2.2.1 Scheduler

A **scheduler** Sch is an algorithm or function that determines the sequence of actions (i.e., a schedule) for executing a concurrent program P . Formally, given a concurrent program P with a transition system $\mathcal{T}_P = \langle S, Act, \rightarrow, s_{init} \rangle$, the scheduler Sch produces a **valid schedule** π , where $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{n-1}} s_n$, satisfying:

1. $s_0 = s_{init}$, the initial state of the program.
2. For all i where $0 \leq i < n$, $\alpha_i \in enabled(s_i)$ and $s_i \xrightarrow{\alpha_i} s_{i+1}$ are valid transitions.
3. $enabled(s_n) = \emptyset$ or Sch terminates at s_n based on its stopping condition.

Algorithm 1 presents a simple randomized scheduler. This scheduler constructs a valid schedule π by iteratively selecting actions α from the set of enabled actions $enabled(s)$ at each state s .

Algorithm 1: A randomized scheduler.

```
1 Function Scheduler (Program P) :  
2    $\pi \leftarrow \text{InitState}(P)$   
3   while  $\text{enabled}(s) \neq \emptyset$  do  
4      $\alpha \leftarrow \text{PickRandomly}(\text{enabled}(s))$   
5      $\pi \leftarrow \pi \cdot \alpha$   
6   return  $\pi$ 
```

Algorithm 2: A concurrency testing algorithm.

```
1 Function ConcurrencyTesting (Program P, Scheduler Sch, and assertions A) :  
2   repeat  
3     // Sample a new schedule  $\pi$ .  
4      $\pi \leftarrow \text{Sch}(P)$   
5     // Verify the schedule satisfies assertions.  
6     for  $a \in A$  do  
7       if  $a(\pi) \neq \text{true}$  then  
8          $\text{raise error}(\pi, a)$   
9   until out of time budget
```

2.2.2 Concurrency Testing Algorithm

A concurrency testing algorithm is a method used to explore the behavior of a concurrent program P under various thread schedules, aiming to detect violations of correctness properties specified by a set of assertions A . It operates by repeatedly generating schedules π using a scheduler Sch , executing P under these schedules, and verifying whether all assertions in A hold for each execution, as shown in Algorithm 2.

2.3 Evolution in Concurrency Testing Platform

Beginning in the 2000s, practitioners introduced stress-testing frameworks that execute test code in highly concurrent environments and repeatedly run tests in an attempt to explore diverse thread scheduling scenarios [1, 2]. These tests are straightforward to implement and compatible with all types of Java applications, regardless of the concurrency primitives employed. Due to their generality and ease of use, stress testing remains one of the most widely adopted methods for developers to evaluate concurrent programs.

Around the same time, researchers explored more controlled and systematic approaches for testing concurrent programs. Their objective was to insert delays and additional synchronization mechanisms to enforce specific happens-before relationships within the program. Tools such as CalFuzzer [36], IMUnit [33], and Thread-Weaver [28] were developed to identify concurrency bugs by selectively influencing event orderings to expose potential issues. These techniques are not designed to enforce whole-program concurrency control. Instead, they target specific event sequences that are likely to reveal bugs. For example, IMUnit and Thread-Weaver rely on

user-provided annotations to define event dependencies, while CalFuzzer employs static analysis to identify error-prone happens-before relationships.

From 2015 to the present, researchers and practitioners have shifted their focus towards whole-program concurrency control to enable more systematic testing. Their approach is to replace concurrency primitives with customized implementations to achieve concurrency control. Java Path Finder (JPF)[71], and LinCheck[37] are two actively maintained concurrency testing frameworks. JPF simulates program execution through a custom bytecode interpreter. However, JPF relies on hand-written implementations of all native JDK library methods. As the JDK evolves, the absence of support for new features limits JPF’s applicability to modern programs.

Lincheck, on the other hand, uses JVM bytecode instrumentation to replace concurrency primitives with custom implementations. While effective for testing the linearizability of concurrent data structures, Lincheck is narrowly tailored to this use case. It requires hand-written specifications of the data structure APIs and supports only a fixed number of concurrent threads. This limitation prevents its use with arbitrary programs that employ custom thread management.

Research tools such as JMVx[63] and JESS[67] also aim to provide general-purpose controlled concurrency testing. Similar to LinCheck, these tools rely on replacing concurrency primitives with custom implementations. JMVx achieves this by utilizing Unsafe APIs to modify concurrency primitives directly. However, this reliance on Unsafe APIs limits its applicability to Java 8, as these APIs were deprecated and subsequently removed in later Java versions.

JESS employs a more aggressive approach by replacing all concurrency primitives and introducing method doubling to avoid interference with the JVM runtime, ensuring that only the application runs the instrumented code. While innovative, this method is inherently fragile, as it cannot handle class constructors, reflections, and virtual methods. As noted by its authors, these limitations frequently result in VM crashes and unpredictable behavior [67].

2.4 Evolution in Testing Algorithms

Despite the inherent challenges in developing a concurrency testing framework for general-purpose concurrent programs, JPF is specifically designed to perform model checking for such programs [71]. Its primary objective is to enumerate all possible schedules of a concurrent program and verify their correctness. CHESS is another tool that aims to perform systematic testing for concurrent programs that use Win32 API. While JPF and CHESS are useful in verifying small concurrent data structures and algorithms, it is hard to apply model-checking in practice as the search space increases exponentially with respect to the size of the application.

As programs grow larger and more complex, researchers and practitioners have increasingly shifted towards testing-oriented approaches, which prioritize identifying bugs over providing formal correctness guarantees. Among these, randomized search-based bounded concurrency testing (RSBT) algorithms have proven to be particularly effective in uncovering concurrency bugs. The key idea behind RSBT algorithms is to randomly sample schedules within a constrained search space that is more likely to expose bugs. Examples of this approach include Probabilistic Concurrency Testing (PCT)[16] and Partial Order Sampling (POS)[78].

The reduced search space is defined by *scheduling constraints*, which are rules that the testing algorithm follows to generate valid schedules. Specifically, probabilistic concurrency

Table 2.1: Comparison of representative distributed-system testing techniques. The key distinction is which sources of nondeterminism each approach can directly control while remaining efficient enough for systematic exploration.

Approach	Representative systems	Threads	Messages	Time	Efficiency
Network interception	FlyMC [50], SAMC [41], Mallory [51]	✗	✓	✗	High
System proxy	MODIST [74]	✓	✓	✗	Low
Deterministic hypervisor	Antithesis	✓	✓	✓	Low

testing (PCT) [16] limits the number of interleavings in the schedule to a small constant, with the observation that this bound is sufficient for reproducing most bugs. Partial order sampling (POS) utilizes the concept of partial order reduction [26], where multiple schedules can be mapped to the same equivalence class of observable behaviors, dramatically reducing search space. These RSBT algorithms have been successfully adopted in several testing tools [16, 17, 19, 54] and have helped uncover real-world bugs in concurrent [16, 54] and distributed systems [17, 77].

Another line of work targets domain-specific bugs, such as data races, deadlocks, or thread-safety violations [34, 35, 44, 76]. For example, TSVD dynamically identifies operation pairs that may lead to thread-safety violations [44], while Razzer directs thread scheduling to uncover kernel data races [34]. While such techniques are useful for finding specific bugs, they are not applicable to a general-purpose testing framework.

Periodical scheduling [73] is a new concurrency testing technique that parallelizes threads in defined periods instead of searching over fixed sequences of interleavings. This is achieved through Linux’s deadline CPU scheduler, which is a system-level control.

2.5 Distributed System Testing

Table 2.1 summarizes the prior work for distributed system testing. Existing approaches differ primarily in *where* they interpose on execution. That choice determines which sources of non-determinism they can control directly, how much semantic visibility they retain into the target application, and how much overhead they impose during testing.

Network interception. Systems such as FlyMC [50], SAMC [41], and Mallory [51] intercept network-facing APIs, RPC libraries, or message-delivery paths and then explore different message orderings, failures, and fault injections. This style of testing is attractive because it is lightweight and scales to large distributed applications. It is particularly effective when the target bug can be exposed by varying message reorderings or node failures while leaving each node’s internal execution mostly unchanged.

The limitation is that message-level control captures only part of the distributed concurrency problem. These approaches typically treat each node as a black box. As a result, they provide little or no direct control over thread interleavings within a node, and their treatment of time is usually limited to network-level delays. Bugs that depend on a precise interaction between local

thread scheduling, message delivery, and time-related failures can therefore remain outside the explored space.

System proxies. MODIST [74] moves lower in the stack by inserting a proxy between the distributed application and the operating system. This gives it broader control than pure network interception: it can influence both message behavior and some concurrency-related events visible at the system boundary. However, MODIST still lacks precise control over time-dependent behavior such as timeouts and timer-triggered events. Besides, this approach becomes less effective when testing distributed systems implemented in managed languages such as Java, where the JVM interposes its own threading and synchronization model on top of OS primitives.

Deterministic hypervisors. At the other extreme, deterministic hypervisor-based systems such as Antithesis aim to control the entire execution environment, including the operating system and the application running above it. This yields a very strong form of control: in principle, threads, communication, and time can all be made reproducible and steerable. Such systems are powerful for debugging and replay, especially when the target software stack is opaque or spans multiple languages and components.

The cost is that the testing mechanism now operates far below the abstractions in which distributed protocols are written. The system must faithfully emulate or control many behaviors that are irrelevant to the application-level property under test. This extra realism often translates into substantial runtime cost and reduced testing throughput. For managed-language systems, the mismatch is more significant: the hypervisor directly controls the runtime and the OS, not the synchronization abstractions that the application developer actually uses.

Chapter 3

FRAY: An Efficient General-Purpose Concurrency Testing Platform for the JVM

The material in this chapter is based on previously published work [42].

As discussed in Section 1.2.1, our first objective is to build a concurrency testing platform that can deterministically execute JVM programs along controlled thread schedules. Ideally, such a framework would (a) provide an efficient mechanism for *controlled concurrency* [68]; that is, to deterministically execute a multi-threaded program along a fixed *schedule* (i.e., sequence of thread interleavings), (b) provide support for systematically or randomly exploring thread schedules using state-of-the-art search strategies (e.g., *partial order sampling* [78]) to uncover hard-to-find concurrency bugs, and (c) target any multi-threaded program without requiring much manual effort (e.g., in rewriting application code or using specialized testing DSLs). Unfortunately, no such general testing framework for the JVM currently exists, despite decades of research on concurrency testing *algorithms*. Why is that?

To understand this gap, we first look to systems described in prior work and map the design space, identifying the subtle trade-offs which make concurrency control of JVM programs challenging. Classical design choices include intercepting OS-level thread-scheduling decisions (as in rr [56], the record-and-replay tool for Linux), by emulating the JVM completely (as in Java Path Finder (JPF) [71]), by mocking concurrency primitives in the JDK (as done by Lincheck [37], for testing concurrent data structures), or by pausing individual threads to defer their scheduling (as in CalFuzzer [36]). As shown in Fig. 3.1, these design choices have impacts on (i) the *scope of what can be tested*, owing to *applicability* (i.e., the extent to which arbitrary target programs are supported), the *expressibility* of the search space (i.e., whether all interleavings can be deliberately and faithfully exercised), and the *maintainability* of the tool itself (i.e., how easily it can keep up with evolving Java versions); as well as (ii) its *bug-finding effectiveness*, which for a given search algorithm (e.g., random walk) depends on maximizing the run-time *performance* of executions and minimizing the *search space* of which interleavings need to be considered.

Crucially, we note that the vast majority of prior academic work in this area has primarily focused on evaluating specific testing techniques or search algorithms, not on maximizing the practical applicability of their artifacts, which has historically been overlooked as an engineering concern. While understandable from a scientific point of view, a side effect is that the gap between

research and practice is wider than ever before, with systems presented at OOPSLA 2024 [63] being limited to Java versions that were superseded in 2017. In contrast, this chapter explicitly investigates the research question: “*Why is concurrency control with managed runtimes challenging, and what are the fundamental system design requirements for maximizing applicability to arbitrary JVM targets?*”

Our key insight is that: a concurrent program does not need to run concurrently to exhibit concurrent behaviors. The observable behaviors of a multithreaded program are determined entirely by which thread interleavings are explored, not by whether threads actually execute in parallel. This means we can serialize a concurrent execution, choosing at each scheduling point which single thread runs next, and still observe every behavior that the original parallel execution could produce.

With this insight, the key design philosophy we establish is to orchestrate thread interleavings (i) without replacing existing concurrency primitives with mocks, while (ii) still encoding the semantics of these concurrency primitives for faithfully expressing the set of all possible program behaviors.

In this chapter, we present FRAY, the first concurrency testing platform for the JVM designed explicitly to maximize both general-purpose applicability as well as application-level bug-finding efficiency, while also providing correctness guarantees and a framework for extensibility. FRAY’s objective is to find concurrency-induced assertion violations, run-time exceptions, as well as deadlocks, in programs that are otherwise testable.¹ To perform concurrency control, FRAY introduces a mechanism called *shadow locking*, which mediates access to shared resources in a specified order (the “thread schedule”) with extra locks whose semantics are coupled to concurrency primitives used in the original program.

FRAY works on off-the-shelf JVM programs (compiled from Java, Scala, Kotlin, etc.), requiring no manual annotation or source rewriting. FRAY efficiently performs deterministic concurrency control over the space of application-thread interleavings where context switches occur only at synchronization points. FRAY’s search space is *sound* and—in the absence of data races and other sources of non-determinism (e.g., timers)—also *complete*; that is, every concurrency bug discovered by FRAY will be a true positive, and for every concurrency bug that can manifest in the original program there is a corresponding interleaving that FRAY can execute to uncover it. In order to search for concurrency bugs, FRAY can run various well-known algorithms such as random walk, probabilistic concurrency testing (PCT) [16], and partial-order sampling (POS) [78]. For debugging purposes, any saved interleaving can be deterministically replayed as long as the program does not depend on other sources of non-determinism.

We also demonstrate FRAY’s push-button applicability to 2,664 concurrent test cases from mature software projects such as Apache Kafka [6], Apache Lucene [7], and Google Guava [5]—we believe this is the *largest evaluation of controlled concurrency testing on real-world software*. FRAY has successfully identified 19 distinct concurrency bugs across these projects, including both *previously unknown* bugs as well as *known* bugs that the developers could not previously reproduce for debugging.

FRAY benefits both practitioners as well as the research community. For example, *Elastic*

¹That is, the programs have entry points or test harnesses for executing logic that does not heavily depend on external sources of non-determinism such as randomness, timing, or networked I/O.

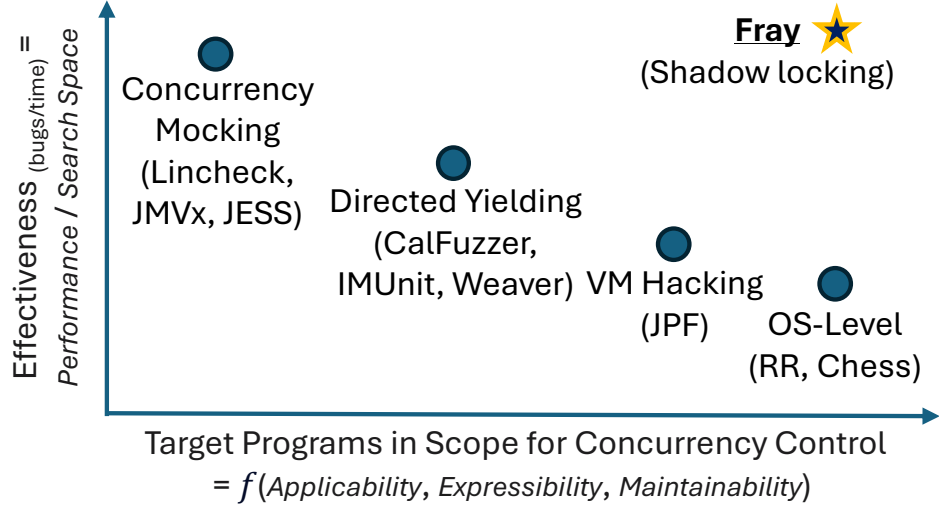


Figure 3.1: FRAY introduces a new abstraction level for concurrency control called *shadow locking*, which identifies a sweet spot in the trade-off between performance overhead, the size of the scheduling search space, and general-purpose applicability.

Search Labs published a blog post about how FRAY helped them diagnose and fix tricky concurrency bugs in Lucene [38]. These bugs were exposed by running existing off-the-shelf unit tests with sophisticated partial-order sampling (POS) [78], which to our knowledge has not been applied at this scale before.

3.1 Design of FRAY

In this section, we describe FRAY, a new platform for concurrency testing of JVM programs designed to meet the first objective (ref. Section 1.2.1) effectively.

Overall architecture FRAY is designed to work on arbitrary JVM programs as a drop-in replacement for the `java` command. FRAY instruments the JVM bytecode of the target program on-the-fly and injects its own run-time library, which spawns a separate *scheduler thread*. Depending on provided input parameters, FRAY either (a) executes the program many times using a provided search strategy (e.g., random, PCT [16], POS [78]) and, if a bug is encountered, outputs a file containing the *thread schedule*, or (b) replays a single program execution with a given thread-schedule file for debugging purposes.

Assumptions and Guarantees FRAY makes two important assumptions to provide soundness and completeness guarantees, but can still be used if these assumptions are relaxed. First, FRAY assumes that the only source of non-determinism in the program is due to concurrency; that is, the program should not make use of `Random`, system I/O, or any timer (e.g., `Thread.sleep`)—this is reasonable for test suites. If true, FRAY guarantees that its search is *sound*; that is, every

discovered “bug” corresponds to an actual behavior that can manifest in the original program. Second, FRAY assumes that programs are free of *data races* (ref. Section 1.1) to guarantee that every concurrency bug can be reproduced by some thread schedule in its search space (though there is no guarantee that a particular search will find it). If the target program contains data races, FRAY can still be used but its search space might exclude some theoretically observable behaviors.

3.1.1 Concurrency Control in FRAY

Our key insight behind designing FRAY is identifying a sweet spot in the abstraction level for concurrency control (ref. Fig. 3.1). FRAY neither manipulates thread management at the operating system level nor replaces concurrency APIs at the language level. Instead, FRAY uses a novel protocol called *shadow locking* to ensure that only one application-level thread executes at a given time and to allow FRAY to control the order in which threads interleave at synchronization points.

Shadow Locking FRAY instruments program classes to add extra synchronization around (but not to replace) concurrency primitives. Fig. 3.2 demonstrates how FRAY would instrument class `Foo` from Fig. 2.1 with additional lock operations.

A *shadow lock* $\mathcal{S}_r^t$ is a lock associated with thread t and resource r . Whenever a thread t wants to access resource r , it must first acquire $\mathcal{S}_r^t$. Shadow locks can control access to the monitor of an object o , denoted as $\mathcal{S}_{\text{monitor}(o)}^t$; access to a volatile variable or atomic value v , denoted as $\mathcal{S}_{\text{volatile}(v)}^t$ or $\mathcal{S}_{\text{atomic}(v)}^t$ respectively; or the permission to start thread execution in the first place, denoted as $\mathcal{S}_{\text{run}}^t$ (for all application threads except the `main` thread).

Shadow locks are instantiated by FRAY dynamically as needed, and immediately acquired by FRAY’s scheduler thread so that no application thread can acquire them by default. FRAY tracks the application’s shadow lock operations (such as `lock` and `unlock`) in order to maintain metadata about threads and their ownership of—or attempts to acquire—shadow locks.

Initially, only the main thread is running and it owns no shadow locks. If a thread is running, it will continue to run until it terminates or attempts to acquire a shadow lock (or executes `Object.wait()`, details later). In the steady state, FRAY maintains the invariant that at most one application thread can be running and all other application threads (if any) are blocked waiting to acquire a shadow lock (or be notified with a signal, details later). When no threads are running, FRAY’s scheduler picks a next thread t to run based on a specified thread *schedule* (i.e., sequence of threads $t_1, t_2, \dots$ to interleave) and releases the shadow lock $\mathcal{S}_r^t$ that t is waiting to acquire. FRAY uses its internal meta-data to ensure that it will only schedule a thread t if it can be guaranteed that t will make progress (i.e., it can acquire the resource r). If no such schedulable t exists, then a *deadlock* is reported. When an application thread releases a shadow lock, the Fray scheduler immediately re-acquires it.

To understand how FRAY instruments Java programs to insert shadow locks, see Figure 3.2, which is the instrumented version of the example from Fig. 2.1, with FRAY’s changes highlighted. *Thread start:* Lines 7 and 33 are inserted to introduce the shadow lock $\mathcal{S}_{\text{run}}^t$ for controlling thread execution, so that new threads are blocked immediately after `Thread.start()` until FRAY determines that no other thread is running. *Re-entrant locks:* Lines 13 and 27 wrap a shadow lock $\mathcal{S}_{\text{monitor}(o)}^t$ around the `synchronized(o)` block (Line 14). A similar logic applies when

```

1  class Foo extends Thread {
2      static Object o = new Object();
3      static AtomicInteger a = AtomicInteger();
4      static volatile int b;
5      public void run() {
6          // Let t = Thread.currentThread()
7           $\mathcal{S}_{run}^t$ .lock()
8
9           $\mathcal{S}_{atomic(a)}^t$ .lock()
10         int x = a.getAndIncrement();
11          $\mathcal{S}_{atomic(a)}^t$ .unlock()
12
13          $\mathcal{S}_{monitor(o)}^t$ .lock() // monitorenter
14         synchronized(o) {
15             if (x == 0) {
16                 int k =  $\mathcal{S}_{monitor(o)}^t$ .getHoldCount();
17                  $\mathcal{S}_{monitor(o)}^t$ .unlock() × k times
18                 do {
19                     o.wait();
20                 } while (!  $\mathcal{S}_{monitor(o)}^t$ .tryLock());
21                  $\mathcal{S}_{monitor(o)}^t$ .lock() × k-1 times
22
23             } else {
24                 o.notifyAll();
25             }
26         }
27          $\mathcal{S}_{monitor(o)}^t$ .unlock() // monitorexit
28
29          $\mathcal{S}_{volatile(b)}^t$ .lock()
30         b = x;
31          $\mathcal{S}_{volatile(b)}^t$ .unlock()
32
33          $\mathcal{S}_{run}^t$ .unlock()
34     }
35     void main() { ... /* see Figure 2.1 */ ... }
36 }

```

Figure 3.2: FRAY’s instrumentation (highlighted) of the program depicted in Fig. 2.1, demonstrating *shadow locking* for concurrency control. Each $\mathcal{S}_*^t$ is a lock instantiated dynamically by FRAY and originally held by FRAY’s scheduler thread; the shadow lock will be released by FRAY when it wants to schedule thread t .

a program uses `ReentrantLock` or `Semaphore` classes. *Atomic and volatile:* The atomic increment at Line 10 and the volatile memory access at Line 30 both propagate information across threads [29, 58]. FRAY conservatively treats these as synchronization points, inserting shadow locks $\mathcal{S}_{atomic(a)}^t$ (at Lines 9 and 11) and $\mathcal{S}_{volatile(b)}^t$ (at Lines 29 and 31) respectively around these accesses.

Handling wait and notify (and why FRAY’s solution is unique) In Java, the semantics of `wait/notify` make controlling concurrency non-trivial. `Object.wait()` can only be invoked by a thread holding a (possibly nested) lock on the object’s monitor, and this call releases the lock (to full nesting depth) and blocks the thread. Correspondingly, `Object.notify()` can only be invoked by a thread holding a lock on the object’s monitor, and when this lock is eventually released the JVM can resume any one thread that was blocked `wait`-ing on this object.

Similarly, `Object.notifyAll()` marks all corresponding wait-ing threads as ready to be woken up by the JVM. The waiting thread(s) must have been in the middle of a (possibly nested) synchronized block; the lock is thus regained (to its original nesting depth) upon waking.

Concurrency control of `wait/notify` is hard because there is no way to control which thread(s) the JVM will wake up or to deterministically replay its choice later. Prior solutions either re-implement the entire JVM (e.g., JPF)—which introduces performance overhead and technical debt—or replace the call with custom implementations (e.g., Lincheck)—which limits applicability, since the JVM can implicitly call `wait/notify` on objects under the application’s control (e.g., when a `Thread` terminates).

FRAY solves this problem using shadow locks as follows. Calls to `o.wait()` (e.g. Line 19 in Fig. 3.2) are (1) preceded by a full release of the shadow lock corresponding to `o`’s monitor (Lines 16–17), (2) wrapped by a loop (Lines 18–20) which only exits when the JVM wakes up the thread and FRAY also makes the shadow lock available. If the shadow lock is unavailable (i.e., `tryLock()` at Line 20 returns `false`), the thread loops back into the `wait()`. If available (i.e., `tryLock()` returns `true`), the loop exits, and the thread regains the shadow lock to its original nesting depth (Line 21). Moreover, (3) FRAY indefinitely holds the thread lock $\mathcal{S}_{\text{monitor}(o)}^t$ until the application calls `o.notify`, and FRAY deterministically selects thread t to be notified.

FRAY also changes each invocation of `o.notify()` to `o.notifyAll()` (Line 24). This ensures that, instead of the JVM picking one arbitrary thread to resume, all waiting threads are woken, but only one thread t , chosen by FRAY, continues execution (by making only its corresponding shadow lock available) while others return to a waiting state.

These semantics are equivalent to the original program! Note that if FRAY cannot or does not want to make the shadow lock for t immediately available (e.g., because some other thread holds the corresponding monitor lock or because we want to explore a different interleaving), then all threads will loop back into the `wait` and FRAY can internally invoke `notifyAll` at a future scheduling step to simulate a delayed wake-up.

If the original program had a call to `notifyAll`, FRAY remembers the number of waiting threads (say w) and internally calls `notifyAll` w times to wake each waiting thread using a deterministic sequence of shadow lock releases to control their waking order. This scheme necessarily introduces a performance cost, but it is not significant enough to slow down FRAY relative to other approaches.

Why Shadow Locks are Better? The primary goal of shadow locks is not to avoid the effort of reimplementing the semantics of concurrency primitives. Instead, shadow locks aim to keep the original concurrency primitives in the program while achieving controlled concurrency. FRAY employs a non-intrusive instrumentation technique that preserves these primitives unmodified, inserting additional code logic around them.

Although in Figure 3.2 it may appear that modifications to `o.notify` and `o.wait` are intrusive, they can actually be implemented non-intrusively. This is achieved by appending an `o.notifyAll` after the original `o.notify` method and transforming the `do...while` loop into a form that appends a `while` loop after the original `o.wait` statement.

This approach ensures that while instrumenting JDK classes that are also used by the language runtime and system threads, their concurrency primitives remain intact and operate as originally

designed. For JVM runtime and system threads, the instrumented code effectively acts as no-ops or empty function calls, and FRAY only activates the instrumentation for the application code, avoiding interference with the JVM’s internal processes while providing controlled concurrency for the application logic.

Other concurrency primitives FRAY handles `Condition.signal/await` similar to `wait/notify`. The JDK implements `Thread.join()` using `Thread.wait()`, where the JVM calls `Thread.notifyAll()` upon thread termination; so, FRAY does not need special handling. FRAY does instrument calls to `Unsafe.compareAndSwap*`, just like atomic/volatile accesses, as well as `Thread.interrupt()`, whose details we omit here for brevity.

Support newer versions of Java The design of FRAY is theoretically compatible with all versions of Java. Our evaluation demonstrates this by successfully testing programs across a wide range of Java versions, from Java 6 to Java 21. Unlike prior approaches, FRAY’s design and implementation do not depend on unsafe APIs or specific JVM internals, making it robust to changes in JVM design or implementation. Instead, FRAY relies solely on an understanding of the concurrency primitive APIs. However, significant changes, such as transitioning the implementation of these primitives from Java to native code, could potentially affect FRAY’s functionality.

3.1.2 Soundness and Completeness

Soundness

We claim that every concurrency bug encountered by FRAY can manifest in the original program if the only source of non-determinism is thread interleaving. To see why, note that FRAY only changes the original program by adding shadow-locking instrumentation as shown in Figure 3.2. Adding extra synchronization does not change program semantics, though it can introduce new deadlocks. We claim that FRAY’s execution deadlocks only if the original program could also deadlock, and this follows from how FRAY couples every shadow lock S_r^t to the program’s attempts to acquire the underlying resource r . Apart from synchronization, FRAY also inserts a `do-while` loop for `Object.wait()` and changes `Object.notify()` to `notifyAll()`, but these are equivalent to the original semantics. We relax our soundness guarantee if the original program uses `wait(timeout)`, which we replace with an indefinite `wait()` to avoid timing-based non-determinism. For such programs, FRAY might report spurious deadlocks if a waiting thread is never woken up with a `notify()`, even though the original program would have made progress after the timeout.

Completeness

We have formally modeled a subset of Java operational semantics to prove FRAY’s completeness guarantee. Using these semantics, sequentially consistent executions of the original program can be represented as *traces*, which are sequences $s_0 \xrightarrow{t_0, e_0} s_1 \xrightarrow{t_1, e_1} s_2 \dots$, where each s_i is a program state, and each $\langle t_i, e_i \rangle$ represents a state transition to s_{i+1} by executing an instruction e_i from

thread t_i . An instruction can either be a synchronization instruction (e.g., `monitorenter` or `atomic`) or a thread-local instruction (e.g., `read` or `branch`), depending on whether or not it creates a happens-before relationship as per the Java memory model [29]. Let $next(s, t)$ be the instruction that thread t would execute if scheduled at state s , or be `undef` if the thread t cannot run. Then, we define a *sync-point-scheduled* (SPS) trace which is a special kind of trace where $t_i \neq t_{i+1} \Rightarrow next(s, t_i) \in \{\text{undef}, \text{monitorenter}, \text{wait}, \text{atomic}, \dots\}$; that is, thread t_i yields control to another thread only if it is blocked/terminated or about to execute a synchronization instruction. FRAY’s shadow locking protocol produces SPS traces. We can then state the completeness theorem:

Theorem 10. *For every assertion violation (evaluated using a predicate made up of thread-local instructions) or deadlock that manifests in a trace π , there exists an SPS trace π' that exhibits the corresponding assertion violation or deadlock.*

Proof. (Sketch) Consider any program trace $s_0 \xrightarrow{t_0, e_0} s_1 \xrightarrow{t_1, e_1} s_2 \dots s_n$. By the assumption that programs are data-race-free, we can swap any pair of consecutive transitions $s_i \xrightarrow{t_i, e_i} s_{i+1} \xrightarrow{t_{i+1}, e_{i+1}} s_{i+2}$, where $t_i \neq t_{i+1}$ and e_i, e_{i+1} are not synchronization instructions, to get a new valid subtrace $s_i \xrightarrow{t_{i+1}, e_{i+1}} s'_{i+1} \xrightarrow{t_i, e_i} s''_{i+2}$ such that $s_{i+2} = s''_{i+2}$. In other words, thread-local instructions across different threads are independent and can be reordered without affecting the resultant state. We can thus perform a series of such reorderings on a program trace exhibiting a concurrency bug until we get (a prefix that is) an SPS trace exhibiting the same bug. $\square$

Finally, we can show that FRAY’s shadow locking protocol can enumerate *all* SPS traces, since FRAY is allowed to release a shadow lock for any thread that is able to acquire the underlying resource. Thus, FRAY’s search space is *complete* under the data-race-free assumption.

3.1.3 Scheduling and Search Strategies

Once we achieve full concurrency control—that is, we can choose arbitrary *schedules* to deterministically execute specific thread interleavings—we can now hunt for concurrency bugs by running *search strategies*. In FRAY, search strategies are plug-ins that implement the logic for choosing the next schedule to run, given some prior search state. The schedules are instantiated dynamically; that is, the program executes a single thread until it is blocked (e.g., when attempting to acquire a shadow lock). Then, FRAY makes a callback to the search strategy, providing the set of *enabled* threads (that is, threads which FRAY knows will make progress if scheduled). The search strategy then returns the next thread to execute.

The search strategies can either be systematic (e.g., depth-first search) or randomized. We do not invent any new algorithms; rather, we look to prior work for picking good strategies. Thomson et al. [68] empirically studied several concurrency testing algorithms including systematic depth-first search, iterative context bounding [53], iterative delay bounding [23], probabilistic concurrency testing (PCT) [16], and the coverage-guided Maple algorithm [75]. They found that PCT was most effective, followed by a naive random walk. Subsequently, Yuan et al. [78] invented the partial-order sampling (POS) algorithm that was shown to outperform PCT.

So, we currently support the following strategies in FRAY: (1) **Random**—At each scheduling step (i.e., callback), pick one of the enabled threads uniformly at random. (2) **PCT**—randomize

Table 3.1: Results of concurrency testing on real-world software. “Tests run” represents the number of test cases that can be executed and whose thread interleavings searched over without internal tool errors. The second column shows the number of test failures identified by each technique (and the subset of failures that are caused by wall-clock-based timed waits). The last column presents the total number of reported bugs (along with the subset confirmed by the developers).

	Technique	Test Run (10 mins each)	Test Failures (Time-related)	Reported Bugs (Confirmed)
Kafka-Stream	FRAY-PCT3	279	126 (60)	6 (6)
	FRAY-POS	279	162 (66)	10 (9)
	FRAY-SURW	279	165 (90)	8 (8)
	FRAY-Random	279	92 (10)	8 (8)
	JPF-Random	0	0 (0)	0 (0)
	rr-Chaos	278	4 (1)	2 (2)
Lucene	FRAY-PCT3	1186	3 (3)	0 (0)
	FRAY-POS	1186	7 (3)	4 (3)
	FRAY-SURW	1186	4 (4)	0 (0)
	FRAY-Random	1186	3 (3)	0 (0)
	JPF-Random	0	0 (0)	0 (0)
	rr-Chaos	1179	0 (0)	0 (0)
Guava	FRAY-PCT3	1199	194 (135)	3 (3)
	FRAY-POS	1199	194 (135)	3 (3)
	FRAY-SURW	1199	199 (137)	3 (3)
	FRAY-Random	1199	196 (137)	3 (3)
	JPF-Random	0	0 (0)	0 (0)
	rr-Chaos	1191	1 (1)	0 (0)

priorities for all threads, and, at each step, schedule the thread t with the highest priority; if this is one of d randomly chosen steps (where d is a user-defined *depth* parameter), then set t to the lowest priority. (3) **POS**—randomize priorities for all threads, and, at each step, schedule the thread t with the highest priority; also reassign random priorities for all other threads competing for the same resource that t is about to acquire. Both PCT and POS algorithms provide probabilistic guarantees for discovering bugs if they exist [16, 78].

3.2 Apply FRAY to Real-World Software

We (attempt to) run all tools, including all search strategies of FRAY, on *every* unit test that spawns more than one thread from the suites of Kafka-streams [6], Lucene [7], and Guava [5]—a total of 2,664 distinct test cases (each running a 10-minute randomized search over schedules). To the best of our knowledge, this is the *largest evaluation of controlled concurrency testing on real-world software*.

Table 3.1 shows the results of these experiments. The table is divided into sections representing each of the target projects: Kafka-Streams, Lucene, and Guava. The column “Tests Run” indicates the number of tests that a given tool could run at all in a push-button fashion, without crashing

with an internal error. Note that the tests are *expected* to pass as they have been regularly running in the projects' corresponding CI pipelines. The next column counts the subset of test runs in which the search strategy found a concurrency bug; that is, the test can be shown to fail (by an assertion violation, run-time exception, or deadlock) due to a race condition. All such failures can be deterministically reproduced with a fixed schedule. A subset of these failures depends on the timers used by the tests (most commonly, `Object.wait(timeout)`)—we identify these explicitly since some of the failures may be highly improbable in practice. Failing tests have to be analyzed manually to deduplicate underlying bugs. The last column shows the number of unique bugs reported to (and confirmed by) the developers.

The results show that FRAY excels in both applicability and effectiveness. First, FRAY can run all the real-world software tests in a push-button fashion. Second, even FRAY-Random identifies interleavings that cause failure in 291 tests (vs. only 5 by rr-Chaos). However, both FRAY-POS and FRAY-SURW work better, finding failures in 360+ tests each—requiring an average of 190 and 82 iterations respectively to search for them. JPF cannot run *any* of the 2,664 tests; it always fails with internal errors due to unimplemented native methods or unsupported JDK features.

We manually deduplicated the bugs identified by the FRAY and reported 18 distinct bugs to project maintainers. For each bug, we have reported detailed instructions for reproduction. Developers have confirmed all bugs at the time of writing. Among the 18 bugs we reported, six are caused by atomicity violations, five by order violations [46], one by unhandled spurious wakeups, five by thread leaks, and the remaining one is yet unclassified. Note that some of the tests in which we observed bugs were encountered as flaky failing tests in past issues, but developers had not identified a way to reproduce and determine the root cause. FRAY unlocks the capability to reliably identify and debug such failures.

Chapter 4

FEST: Feedback-guided Adaptive Testing of Distributed Systems Designs

The material in this chapter is based on previously published work [43].

Once we have a mechanism to control the execution of a concurrent program, a natural goal is to test it by systematically exploring all possible schedules. However, the search space is usually too large, and developers often have time and computing constraints. The question then becomes: which schedules should we explore, and how should we choose them?

In practice, developers rely on *randomized testing* [14, 15] to trade exhaustive guarantees for practical scalability. Current state-of-the-art techniques are often variants of *randomized search-based bounded concurrency testing (RSBT)* [16, 77, 78], such as Probabilistic Concurrency Testing (PCT) [16] and Partial Order Sampling (POS) [78]. These algorithms prune the search space using scheduling constraints (e.g., limiting context switches [16] or focusing on partial-order dependencies [78]), focusing exploration on schedules deemed more likely to expose unique behaviors or bugs (visualized by the blue ellipse in Figure 4.1). This constrained approach has proven effective in practice for finding bugs compared to unconstrained random exploration.

However, existing RSBT algorithms suffer from a fundamental limitation: *within* the constrained space, they still rely on random sampling. Randomness, akin to black-box fuzzing [12], is inefficient for maximizing behavioral coverage because it lacks guidance from past executions. It does not learn which types of schedules are more likely to reveal new behaviors or trigger corner-case bugs. Consequently, developers using these tools often lack confidence in the testing thoroughness achieved within their time budgets. While adaptive techniques like Q-learning (QL) [52]—which learn scheduling actions based on system state and rewards—seem a natural solution, they have proven surprisingly ineffective. Our experiments (Section 4.4.3), consistent with prior studies [17], show QL performing even worse than RSBT methods on complex systems. This underscores the value of RSBT’s structural constraints but highlights a missing ingredient: the exploration within these constraints needs guidance from something other than randomness.

Our key insight is derived from the behavior-centric view developed in this thesis: we should let *observed program behaviors* guide the search. Two schedules that produce identical observable behaviors are redundant from a testing perspective, regardless of how structurally different they appear. Conversely, a schedule that produces a previously unseen behavior is valuable even if it

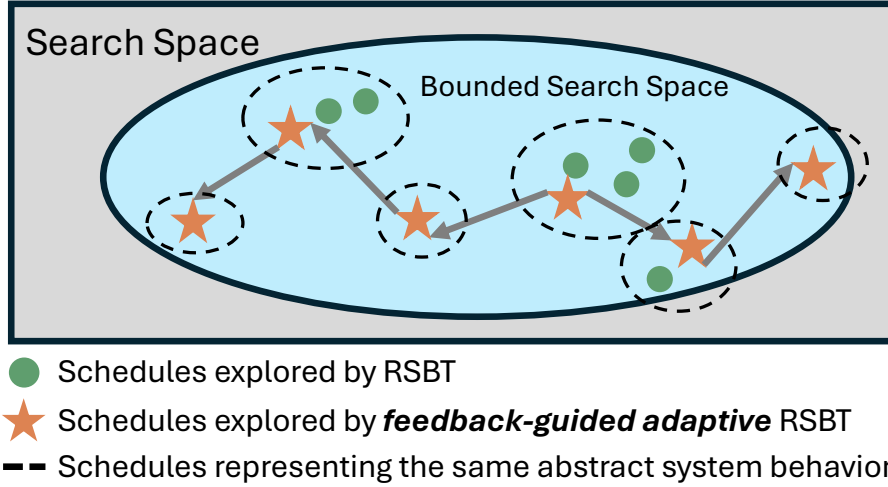


Figure 4.1: Overview of the search performed by RSBT and feedback-guided adaptive RSBT. Each edge connects two schedules, indicating that the target schedule is generated adaptively based on the source schedule.

differs only slightly from a prior schedule. Behaviors, not schedules, are the fundamental unit of testing progress.

Building on this insight, we propose integrating feedback-guided adaptive schedule generation into RSBT algorithms. Inspired by the success of grey-box fuzzing [12] in software testing, the idea is to evaluate each executed schedule by the behaviors it produces, retain schedules that exhibit novel behaviors, and mutate them to generate subsequent schedules that explore nearby but unexplored behavioral regions—all while respecting the beneficial structural constraints of RSBT.

Realizing this idea for distributed system testing presents two key challenges. First, directly mutating a schedule, such as altering the execution order between machines, can produce infeasible schedules (e.g., causing a machine to become unschedulable by blocking it from receiving a necessary message) or violate RSBT constraints (e.g., exceeding the permitted number of context switches). Second, given the vast space of possible schedules, the feedback mechanism must efficiently distinguish schedules that produce genuinely new behaviors from those that merely appear different structurally.

This chapter presents FEST, the first distributed design testing technique aimed at maximizing behavioral coverage. FEST addresses these two challenges as follows. For the first challenge, FEST integrates record-and-replay [56] techniques with parametric input generation [60]: it records the random choices made by an RSBT algorithm during execution and applies small, controlled mutations to these recorded choices. The mutated recordings are then replayed to generate new schedules that are similar yet distinct from the original, ensuring that the resulting schedules remain feasible and respect the algorithm’s constraints. For the second challenge, FEST introduces a *diversity feedback* mechanism that evaluates the behavioral uniqueness of each schedule. This evaluation is based on abstract Lamport timelines [39, 51], which capture the happens-before relationships between system events. Schedules exhibiting higher diversity (i.e., novel happens-before orderings) are prioritized for future mutation and exploration (illustrated by

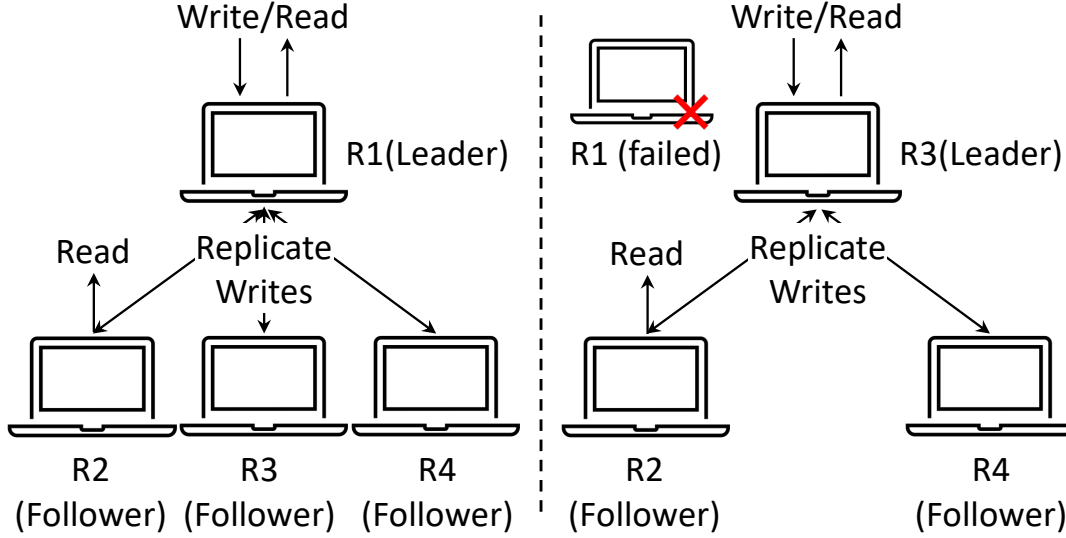


Figure 4.2: A distributed storage system with four replicas.

the arrows and stars in Figure 4.1).

With adaptive schedule generation, developers can also optionally guide the scheduling algorithm to prioritize critical *scenarios*, a specific sequence of events or messages processed by the distributed system. This is important because focusing on critical scenarios ensures that the system’s most significant and potentially vulnerable behaviors are thoroughly tested, increasing the likelihood of detecting faults that may only manifest under specific conditions. To support this, FEST introduces a novel scenario language to guide the algorithm toward desired or otherwise interesting system behaviors. Using this approach, FEST can prioritize generating schedules that align with the scenario specifications.

We integrate the FEST approach into the P programming framework. We evaluated FEST on 5 open-sourced models, including the recently released distributed file system model from DeepSeek [4], and 14 industrial distributed systems models from AWS. The results demonstrate that FEST outperforms existing RSBT algorithms such as PCT and POS as well as Q-learning. Specifically, FEST achieves up to $41\times$ ($1.5\times$ on average) more behavioral coverage, $278\times$ ($15\times$ on average) more scenario coverage, and detects 33% more bugs. Moreover, our evaluation demonstrates that diversity-based priority sampling enables FEST to achieve a $1.22\times$ improvement in timeline coverage.

4.1 Background

4.1.1 P Framework

P [18, 19, 21] is a programming language for modeling distributed systems as communicating state machines. It provides full control over event sequencing, enabling support for various testing techniques, including model checking [18], concurrency testing [19], and symbolic execution [20],

```

1  event readReq: (key: str, rId: int);
2  event readResp: (key: str, val: int, rId: int);
3  event writeReq: (key: str, val: int, rId: int);
4  event writeResp: (status: bool, rId: int);
5  event newLeaderElected: (leader: Replica);
6  machine Replica {
7    var kvStore: map<str, int>;
8    var allReplicas: set[Replica];
9    var currLeader: Replica;
10   ...
11   state Leader {
12     on readReq (req) do {
13       // on read request, look up the local store and
14       // send response to client
15       send req.sender, readResp, (key = req.key, val = kvStore[req.key], rId = req.rId);
16     }
17     on newLeaderElected(nle) do {
18       currLeader = nle.leader;
19       if (nle.leader != this) goto Follower;
20     }
21     on writeReq (req) do {
22       // apply write locally, reliably replicate it and
23       // send response back
24       kvStore[req.key] = req.val;
25       var status = ReplicateWriteReq(allReplicas, req);
26       send req.sender, writeResp, (status = status, rId = req.Id);
27     }
28   }
29   state Follower {
30     on readReq (req) do {
31       // on read request, look up the local store and
32       // send response to client
33       send req.sender, readResp, (key = req.key, val = kvStore[req.key], rId = req.rId);
34     }
35     on newLeaderElected (nle) do {
36       currLeader = nle.leader;
37       if (nle.leader == this) goto Leader;
38     }
39     on writeReq (req) do {
40       // forward the request to correct leader
41       send currLeader, writeReq, req;
42     } } }
43   // Entry machine creates 4 replicas
44   machine TestFourReplica {
45     var replicas: set[Replica];
46     var i: int;
47     start state init {
48       entry {
49         while (i < 4) {
50           replicas += new Replica();
51           i += 1;
52         }
53         ...
54       } } }

```

Figure 4.3: Replica state machine in P.

to verify that the system model satisfies its desired specifications. P is used across industry (e.g., AWS [59], Microsoft [18], and DeepSeek [4]) and academia [20, 21, 22] for reasoning about distributed protocols. At AWS, P is used for reasoning about the correctness of core distributed protocols driving services across compute, databases, and storage systems.

Algorithm 3: The testing loop using an RSBT algorithm.

```
1 Function RSBTesting(Program P, RSBT algorithm A):  
2   repeat  
3     // Sample a new schedule S.  
4      $S \leftarrow A(P)$   
5     // Get execution trace E.  
6      $E \leftarrow \text{Execute}(P, S)$   
7     if E violates specification assertions then  
8       raise error(E)  
9   until out of time budget
```

Replicated storage system modeled in P. We consider a simple replicated key-value store consisting of one leader and three follower replicas (Figure 4.2). The leader replica processes all write requests, and the leader is responsible for reliably replicating these write operations to the other replicas. Read requests can be serviced by any replica, as each replica maintains a consistent version of the replicated data. When the leader fails, a leader-election service is used to elect a new leader and return the system to a stable state (Figure 4.2 right side).

Figure 4.3 presents the P state machine for each `Replica` node. The state machines communicate by sending events, defined between Line 1 and Line 5. The `Replica` machine has two states: `Leader` and `Follower`. In both states, upon receiving a `readReq` event, the replica retrieves the value from its store and responds to the client (Line 15 and Line 33). In the `Leader` state, it handles write requests by applying the write locally (Line 24) and reliably replicating it on all follower replicas (Line 25). If a follower receives a `writeReq`, it will forward the request to the correct leader. We assume a leader election service detects leader node failures, elects a new leader, and informs all replicas of the new leader using `newLeaderElected` events. The replica uses this event to transition between the leader and follower states (Line 19 and Line 37). The main property the key-value store must satisfy is the consistency (linearizability) of write and read operations, ensuring that the distributed system behaves like local storage (e.g., a hash map).

The machine `TestFourReplica` serves as the entry point of the P program, analogous to the main function in C or Java. This machine typically configures the distributed system and generates payloads; in this example, it creates four replicas.

4.1.2 Schedules and Randomized Search-based Bounded Concurrency Testing

The advantage of modeling distributed system designs in frameworks like P is that they can be extensively tested (or explored) by precisely controlling the interleavings of messages sent between various machines—a sequence of machine interleavings is called a *schedule*. The search space of all possible schedules is massively large. So, state-of-the-art techniques for testing system models improve scalability by performing random sampling of schedules within a reduced search space that is more likely to reveal bugs. For example, probabilistic concurrency testing (PCT) restricts exploration to schedules with few context switches, leveraging empirical findings that most concurrency bugs manifest under such conditions [16]. Partial order sampling (POS) applies

Algorithm 4: The deterministic record and replay algorithm given an RSBT algorithm.

```
1 Function Record(Program P, RSBT algorithm A) :  
2   R ← new Recording()  
3   originalRandInt ← RandInt  
4   override Function RandInt(bound b) :  
5     // Getting a random integer from OS.  
6     r ← originalRandInt(b)  
7     R.append(r)  
8     return r  
9   // Running RSBT with overridden RandInt.  
10  S ← A(P)  
11  return S, R  
  
12 Function Replay(Program P, RSBT algorithm A, Recording R) :  
13  override Function RandInt(bound b) :  
14    // Fetching a saved random from recording.  
15    r ← R.next()  
16    return r (mod b)  
17  // Running RSBT with overridden RandInt.  
18  S ← A(P)  
19  return S
```

partial order reduction [26] to avoid exploring equivalent schedules that exhibit identical partial order behaviors.

We call such approaches *randomized search-based bounded concurrency testing* (RSBT). An RSBT algorithm is defined by a function (e.g., `nextSchedule(P)`) that returns a new schedule for the program *P* based on the pseudo-randomness provided by the operating system. These RSBT algorithms have been successfully adopted in several testing tools [16, 17, 19, 54] and have helped uncover real-world bugs in concurrent [16, 54] and distributed systems [17, 77].

Algorithm 3 shows the high-level design of an RSBT-based testing framework. The algorithm takes an input program *P* and an RSBT algorithm *A* as input. In each iteration, the algorithm generates a new schedule *S*, runs the program *P* using this schedule, and records the execution events *E*. If any execution *E* violates a specification assertion, an error is raised and recorded.

4.1.3 Deterministic Record and Replay

Record-and-replay techniques have been successful in debugging complex software [56]. The core idea is to record the non-deterministic events during the execution of a program—such as time, thread interleaving, and random numbers—in such a way that the exact same execution can be deterministically reproduced later.

One key property for the existing RSBT algorithms is that the execution of the algorithm is purely determined by the input program and pseudo-random numbers provided by the operating system. With record-and-replay techniques, these random choices can be recorded during the original execution and replayed exactly to produce the same schedule.

Algorithm 4 describes how deterministic record and replay works for an RSBT algorithm.

Both `Record()` and `Replay()` methods override the original `RandInt` method provided by the operating system. For RSBT, a recording is simply a list of integers. In the recording phase, the algorithm logs all random choices made during the execution of the program (Line 6). In the replay phase, instead of generating new random choices, the algorithm returns the recorded values (Line 12), ensuring A reads the same value. If a schedule S and recording R are produced during the recording phase `Record( $P, A$ )`, the algorithm guarantees that the same schedule S will be faithfully reproduced during the replay phase `Replay( $P, A, R$ )`.

$$S, R = \text{Record}(P, A) \implies S = \text{Replay}(P, A, R)$$

4.1.4 Behavioral Coverage for Distributed Systems

Given the efficient sampling-based approach of RSBT, how does a user gain confidence in whether their system is thoroughly tested (perhaps when the testing tools do not report new bugs)? In sequential program testing, *code coverage* approximates testing thoroughness by measuring the fraction of code exercised by test cases. However, such measurement is insufficient for distributed systems, where *schedules* influence behavior much more than intra-machine control flow. Prior studies have shown that code coverage tends to saturate quickly because the same codebase is deployed across multiple machines [51]. Recently, researchers have proposed to measure behavioral coverage in distributed systems via *abstract Lamport timelines (timeline coverage)* [51]. Timelines capture the happens-before relationships between events or messages exchanged. In this chapter, we use the notion of such timeline coverage as a proxy for *behavioral coverage*; a timeline for a P program is the happens-before relationship between events exchanged by the state machines. Developers can utilize timeline coverage to assess the effectiveness of automated tools in exploring their distributed systems.

4.2 Problem and Solution Overview

4.2.1 Problem Statement

Our main hypothesis is that while RSBT techniques like PCT and POS effectively reduce the search space to find bugs, the sampling-based approach is not adaptive and, therefore, not tailored to maximize timeline coverage. This is analogous to black-box fuzz testing in sequential programs: randomly sampling inputs rarely results in good code coverage [12].

Our goal is to extend *any* existing RSBT approach to achieve better timeline coverage. We accomplish this by incorporating timeline coverage as feedback to RSBT algorithms, allowing them to generate schedules adaptively.

4.2.2 Feedback-Guided Adaptive Scheduling

In this chapter, we aim to combine the feedback-guided mutation-based input generation technique with the approach of sampling schedules in RSBT for distributed systems to achieve high timeline coverage. To achieve this, we need to address two challenges.

Challenge 1: adaptive schedule generation. The first challenge is how to adaptively generate new schedules while maintaining the same search-space bounding as RSBT algorithms like PCT and POS. This requires creating schedules that respond to feedback but still respect the constraints imposed by these sampling techniques. Note that directly mutating a schedule of machine interleavings will often lead to infeasible schedules since not every sequence of machine interleavings is actually realizable in a given system; a mutated schedule is also not guaranteed to remain within the bounded search space imposed by RSBT algorithms. For example, consider a schedule for the replica state machine shown in Figure 4.2. In this scenario, one machine sends a `newLeaderElected` event to the current leader, signaling the election of a new leader and causing the old leader to transition to the follower state. Following this, another machine sends a `writeReq` to the old leader. The old leader, now a follower, correctly forwards the request to the new leader. However, if we mutate this schedule by simply swapping the order of these events—so that the `writeReq` is sent before the new leader is elected—the old leader will handle the `writeReq` without forwarding it. This results in an infeasible schedule, as the new leader never receives the `writeReq`.

Our solution. FEST integrates the concepts of record-and-replay [56] and parametric generators [60]. FEST treats the RSBT algorithm as parametric generators, where the schedule generation process is purely determined by the distributed system model and some pseudo-random choices. A *recording* contains all pseudo-random choices taken by the RSBT algorithm, and replaying the recording given the same RSBT algorithm and program will produce the same schedule. With this capability, FEST can sample a new schedule similar to the recorded one. By *mutating* the recording—instead of strictly adhering to the original recording, FEST introduces deviations by returning different random values during the replay process. Consequently, the new schedule is slightly altered from the previous one. Since the replay process uses the same RSBT algorithm as a generator, the new schedule also satisfies the scheduling constraints. Hence, continuing to leverage the benefits of performing RSBT.

Challenge 2: effective schedule prioritization. Complex distributed systems have a very large scheduling space, where thousands of schedules may correspond to a unique timeline. Exploring all saved schedules with equal priority is both inefficient and time-consuming, as prior research has shown that mutating different saved inputs varies in their likelihood of revealing new coverage [13, 61]. Thus, it is necessary to prioritize the exploration of schedules that are more likely to expose unexplored timelines, thereby maximizing coverage and reducing the time required to uncover unique timelines.

Our solution. FEST assigns a priority to each schedule by evaluating its *diversity feedback*, which measures how unique the new timeline is from those previously explored. A schedule receives a higher diversity score if its corresponding timeline significantly differs from others. By prioritizing the mutation of schedules with higher diversity scores, FEST enhances the likelihood of uncovering new, unique execution paths.

4.2.3 Scenario-Guided Schedule Generation

At AWS, one common request from the developer is to guide the scheduling algorithm towards a specific *scenario*. Scenarios are behaviors of the distributed system that specific schedules can reach. For instance, in the case of the replicated storage store, the following are some scenarios that service teams at AWS wrote, with increasing levels of complexity:

- **S1: Read-After-Write.** The system processes a write request and then handles a read request for the same key.
- **S2: Write then read with a leader change.** A leader node processes a write request for a specified key. Later, a new leader is elected and handles a read request for the same key.
- **S3: Read and write when leader flip-flops.** The scenario starts with a leader node processing a write request for a specific key. A new leader is then elected due to a temporary partition, but as the network recovers, the original leader is reelected and processes a read request for the same key.

Developers are often interested in testing specific scenarios whose correctness may be impacted by their development activities. For example, when implementing a new failure resolution algorithm, developers would typically be interested in checking the system’s correctness under scenarios **S2** and **S3** that involve leader transition. The current state-of-the-art techniques for automated testing, like RSBT, do not provide any mechanism to guide the sampling process toward these scenarios of interest specifically.

Our primary evaluation indicates that popular RSBT algorithms such as PCT and POS achieve poor coverage for many subtle yet critical scenarios. For example, a 1-hour test of AWS’s internal storage protocol using PCT failed to explore any timeline that involved concurrent split-merge operations (SC in Figure 4.8). Consequently, many scenarios remain unexplored for years, which poses significant risks. These unexplored scenarios have led to bugs in our experience.

FEST utilizes the abstract *execution events* to capture the behaviors of distributed systems. The execution events are a sequence of P events (e.g., `readReq` and `writeReq` in Figure 4.3) transmitted during the execution. Given the execution events of a schedule, FEST provides a specification language for developers to define scenarios. We show the detailed design of our specification language in Section 4.3.5. If a scenario is provided, FEST translates these scenarios into constraints used to compute the *compliance feedback*, measuring the extent to which specified scenarios are explored. Next, FEST prioritizes the mutation process of schedules with higher compliance scores. This helps FEST prioritize exploring schedules that are more likely to satisfy the defined scenarios.

4.2.4 Scope and Limitations

FEST only works for stateless RSBT algorithms, meaning that pseudo-random choices and the input distributed systems purely determine the schedule generated by these algorithms. FEST does not support learning-based concurrency testing algorithms such as Q-learning. While FEST is designed and implemented to test distributed system models, the core algorithm can be applied to all concurrent system testing if the testing framework controls the message/thread order.

Algorithm 5: Feedback-guided adaptive scheduling algorithm for distributed system models. Four main components are highlighted in different colors.

```

1 Function Main (Program P, RSBT algorithm A, optional Scenario scen) :
  // Initial recording is a random sample.
2   $S, R \leftarrow \text{Record}(P, A)$ 
3   $\mathbb{R} \leftarrow \{R\}$ 
4   $\mathbb{S}, \mathbb{E} \leftarrow \emptyset$ 
5  repeat
6     $R \leftarrow \mathbb{R}.\text{Dequeue}()$ 
7    repeat
8      // Adaptive Schedule Generation (§4.3.2)
9       $R' \leftarrow \text{Mutate}(R)$ 
10      $S' \leftarrow \text{Replay}(P, A, R')$ 
11     // System Execution (§4.3.3)
12      $E \leftarrow \text{Execute}(P, S')$ 
13     if  $E$  violates specification assertions then
14        $\text{raise error}(E)$ 
15     // Diversity Feedback (§4.3.4)
16      $\text{diversity} \leftarrow \text{Diversity}(\mathbb{E}, E)$ 
17     if  $\text{diversity} = 0$  then
18       continue
19     // Compliance Feedback (§4.3.6)
20     if scen is provided then
21        $\text{compliance} \leftarrow \text{Compliance}(\text{scen}, E)$ 
22     else
23        $\text{compliance} \leftarrow 1$ 
24      $\mathbb{E} \leftarrow \mathbb{E} \cup \{E\}$ 
25      $\mathbb{S} \leftarrow \mathbb{S} \cup \{S'\}$ 
26      $\mathbb{R}.\text{Insert}(R', \text{diversity} * \text{compliance})$ 
27   until out of exploration budget
28 until out of time budget
29 return  $\mathbb{S}$ 

```

4.3 FEST Design

4.3.1 Main Algorithm

Algorithm 5 shows the feedback-guided adaptive scheduling algorithm. FEST takes a program P , an RSBT algorithm A and an optional user-specified scenario scen as input. Upon completion, FEST outputs a collection of schedules, each representing a unique timeline. The algorithm starts by running the RSBT algorithm A with pseudo-randomness values generated by the operating system and saves the recording in a priority queue, $\mathbb{R}$.

During each iteration, the algorithm dequeues a saved recording R with the highest priority (Line 6). For each recording, FEST invokes the `Mutate` method to modify R , followed by the `Replay` method to generate a new schedule S' based on the mutated recording R' . The details of the adaptive schedule generation algorithm are described in detail in Section 4.3.2.

With the new schedule S' , FEST orchestrates the program P and stores the execution events in E (Line 12).

FEST gathers diversity feedback for the new schedule (details of this computation are presented in Section 4.3.4). The `Diversity` function returns an integer *diversity* (Line 16), used to 1) decide whether FEST should save the newly generated schedule S' and its recording R' , and 2) assign a priority to this schedule. If *diversity* is not zero, it indicates a previously unexplored timeline that should be saved for further exploration. Otherwise, the algorithm should not save this schedule and continue (Line 17).

If a scenario is provided (Line 20), the algorithm computes the compliance feedback (Line 21) as detailed in Section 4.3.6. Otherwise, it sets *compliance* to 1 (Line 23), indicating that all schedules have the same compliance score.

Finally, FEST saves the new schedule to $\mathbb{S}$, the new execution event to $\mathbb{E}$, and inserts the new recording R' into the priority queue $\mathbb{R}$ with a priority based on the product of *diversity* and *compliance* (Line 24-Line 26).

FEST allocates a mutation budget proportional to the *diversity* and *compliance* score of each saved schedule—schedules with higher scores receive more mutations—and continues until the budget is exhausted (Line 27).

4.3.2 Adaptive Schedule Generation

A *schedule* S is defined as a sequence of machine interleavings $m_1, m_2, \dots$ where each scheduled machine either sends a message to another machine or receives a message from its buffer. Given a schedule S , the goal of the adaptive schedule generation is to generate a new schedule S' , which shares similarities with S . For two reasons, FEST does not directly mutate schedules in its testing loop. First, we aim to generate schedules satisfying scheduling constraints defined by the RSBT algorithm, as these strategies have been proven more effective in bug detection. Second, not all schedules are valid for executing system P . Given a prefix of interleavings $\bar{S}$, only a subset of available machines may be schedulable immediately after $\bar{S}$. We denote the set of such *enabled* machines when program P is executed with schedule prefix $\bar{S}$ as $\text{Enabled}(P, \bar{S})$. Random point mutations on a schedule will not always result in a valid schedule; changing one scheduling decision in the sequence will affect the validity of every subsequent scheduling decision. So, FEST combines the concept of parametric input generation with deterministic record-and-replay and introduces small mutations while replaying the schedule generation algorithm, as described next.

The key to our mutation algorithm is to mutate recordings instead of mutating schedules; by adding this indirection, FEST controls the generation of the schedule by mutating the corresponding recording. Recall that a recording is represented as a list of integers. Given a schedule S and its corresponding recording R , the `Mutate` method generates a new recording R' by modifying a subset of integers in R . The `Replay` (P, A, R') method then produces a new schedule S' . The degree of similarity between S and S' is governed by the `Mutate` method. If `Mutate` is the identity function, S' will be identical to S . Conversely, if every integer in R is modified, the `Replay` function will generate a completely different schedule, akin to random sampling. In FEST, our objective is to create schedules that are similar but not identical. To achieve this, FEST mutates an average of five integers per recording.

Algorithm 6: The PCT algorithm. The function calls to pseudo-randomness are highlighted in blue.

```

1 Function PCT (Program P) :
2   depth  $\leftarrow$  user configured hyperparameter
3   Q  $\leftarrow$  PriorityQueue(initial = {entryMachine})
4   S, K  $\leftarrow$   $\emptyset$ 
5   for idx  $\in$  {1, ..., depth} do
6     k  $\leftarrow$  RandInt (max schedule length)
7     K  $\leftarrow$  K  $\cup$  {k}
8   repeat
9     // Pick highest priority enabled machine
10    for idx  $\in$  {0, ..., |Q - 1} do
11      if Q[idx]  $\in$  Enabled(P, S) then
12        m  $\leftarrow$  Q[idx]
13        break
14    S  $\leftarrow$  S m
15    // Change the priority of m if a switch point is hit.
16    if |S|  $\in$  K then
17      Q.MoveToTail(m)
18    // Add new spawned machine to Q.
19    for m'  $\in$  Enabled(P, S) do
20      if m'  $\notin$  Q then
21        prio  $\leftarrow$  RandInt (|Q|)
22        Q.Insert(prio, m')
23  until Enabled(P, S) =  $\emptyset$ 
24  return S

```

Probabilistic Concurrency Testing. The core idea behind the probabilistic concurrency testing algorithm (PCT) is that most bugs can be revealed using a small number of context switches. Therefore, given a *bug depth*, the minimum number of context switches needed to trigger a bug, PCT samples schedules with a limited number of thread interleavings by controlling the points at which context switches are introduced through priority-based scheduling.

Algorithm 6 illustrates the PCT algorithm, with the function calls to pseudo-randomness highlighted in blue to emphasize the points where random choices influence the schedule generation. The algorithm uses a hyperparameter *depth* to limit the number of context switches in generated schedules. The algorithm begins by initializing a priority queue *Q*, an empty schedule *S*, an empty set of context switch points *K*, and an empty recording *R'* (Line 3-4). The priority queue *Q* is populated with only the main thread—in our case, the machine containing the entry point—to start the execution. It then samples *depth* context switch points stored in *K* (Line 5-Line 7) by calling RandInt method. At each scheduling point, the algorithm selects the highest-priority machine that is ready to execute (Lines 9-12) and appends it to the schedule (Line 13). When *K* contains the trace length of the current execution *S*, the algorithm lowers the priority of the currently scheduled machine (e.g., by moving it to the tail of the priority queue), thereby introducing a context switch occurring at the next scheduling point (Line 15). Finally, if the schedule enables a

Algorithm 7: The POS algorithm. The function calls to pseudo-randomness are highlighted in blue.

```

1 Function POS (Program P) :
2    $Q \leftarrow \text{PriorityQueue}(\text{initial} = \{\text{entryMachine}\})$ 
3    $S \leftarrow \emptyset$ 
4   repeat
5     for  $idx \in \{0, \dots, |Q| - 1\}$  do
6       if  $Q[idx] \in \text{Enabled}(P, S)$  then
7          $m \leftarrow Q[idx]$ 
8         break
9       // Reset priorities
10      for  $m' \in \text{Enabled}(P, S)$  do
11        if  $\text{IsRacing}(m', m)$  then
12           $Q \leftarrow Q \setminus \{m'\}$ 
13       $S \leftarrow S \cup m$ 
14      for  $m' \in \text{Enabled}(P, S)$  do
15        if  $m' \notin Q$  then
16           $prio \leftarrow \text{RandInt}(|Q|)$ 
17           $Q.\text{Insert}(prio, m')$ 
18  until  $en_P = \emptyset$ 
19  return  $S$ 

```

new machine m' which is not in Q , PCT inserts m' to Q with a random priority (Line 16-19).

Partial Order Sampling Algorithm 7 presents the schedule generation algorithm that employs partial order sampling [78]. At each iteration, the algorithm selects a machine m with the highest priority that is ready to execute (Lines 5 to 8). POS resets the priority of a machine according to partial order reduction rules [26] and removes machines from Q if they access the same resource as machine m (Lines 9- 11). Finally, it adds m to the schedule S (Line 12) and inserts new machines to Q with a random priority (Line 13-16).

Adopt Other Schedule Generation Algorithm. Note that our adaptive schedule generation algorithm is generic and accommodates various RSBT algorithms. For example, our evaluation includes POS+, an extension of the POS [78] algorithm with conflict analysis [77]. Section 4.4.3 presents a comparative analysis of each feedback-guided strategy against its random-sampling-based equivalent, highlighting improvements in both timeline coverage and bug-finding capabilities.

4.3.3 System Execution

Back to Algorithm 5, the next step after generating the schedule of interleavings S is to execute the program P with that schedule S and observe various events being generated by machines during execution (Line 12). To ensure the reliability and correctness of P , we employ runtime monitors within the execution framework. These monitors check system execution against protocol specifications (Line 14). The execution, represented as a sequence of events E , is processed to

Algorithm 8: Diversity feedback algorithm.

```
1 Function ToTimeline (Execution events  $E$ ) :  
2    $T \leftarrow \emptyset$   
3   for  $e, e' \in E$  do  
4     if  $e' \prec e \wedge e'.receiver = e.receiver$  then  
5        $T \leftarrow T \cup \{ \langle e.receiver, e'.name, e.name \rangle \}$   
6   return  $T$   
7 Function Diversity (Execution history  $\mathbb{E}$ , List of Events  $E$ ) :  
8    $\mathbb{T} \leftarrow \bigcup_{E' \in \mathbb{E}} \text{ToTimeline}(E')$   
9    $T \leftarrow \text{ToTimeline}(E)$   
10  return  $\min_{T' \in \mathbb{T}} (1 - J(T, T'))$ 
```

evaluate the feedback of generated schedule S .

4.3.4 Diversity Feedback

Feedback is important in guiding FEST in generating schedules that achieve high timeline coverage. Post execution, FEST processes the event sequence E to extract diversity feedback (Algorithm 5, Line 16). Diversity feedback aims to identify schedules that are more likely to uncover unique behaviors during future exploration. However, directly comparing schedules or sequences of machine interleavings is too detailed. Many different schedules can result in the same high-level program behavior, especially when the differences involve independent local computations that do not affect information exchange between machines. We need a better way to measure differences in higher-level effects between executions.

Inspired by prior work [51], we use abstract Lamport timelines to measure the uniqueness of schedules. The abstract Lamport timeline is defined as $T :: \{ \langle M, e_1, e_2 \rangle \}$, which is a set of tuples each tuple captures a *happens-before* relationship: that on a machine M , an event of type e_1 is received and processed before another of type e_2 [39, 51]. The `ToTimeline` function shown in Algorithm 8 shows how to transform the execution events E to an abstract Lamport timeline. It iterates over pairs of events with the same receiver and creates tuples that capture their happens-before relationship.

Algorithm 8 presents the `Diversity` function, which calculates diversity feedback based on execution history $\mathbb{E}$ and current execution events E . It first generates explored abstract timelines $\mathbb{T}$ from $\mathbb{E}$ (Line 8) and the new abstract timeline T from E (Line 9). To measure diversity between two sets, FEST employs a *diversity index*, defined as $1 - J(T_1, T_2) = 1 - \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|}$, where $J(T_1, T_2)$ is the Jaccard index. The Jaccard index computes the similarity of two sets by measuring the ratio of their intersection to their union. A higher diversity index indicates greater diversity (less similarity) between the sets, while a lower index indicates they are more similar (less diverse). Finally, the `Diversity` function returns the minimum diversity index between the new timeline T and all previously explored timelines $\mathbb{T}$ (Line 10), with zero indicating that T is identical to a timeline T' in $\mathbb{T}$.

$\mathcal{S} := \langle name \rangle (\text{Event}(\cdot, \text{Event})^*) :$ $\mathcal{C} (\wedge \mathcal{C})^*$ $\mathcal{C} := \text{Event}(\prec \text{Event})^+$ $ \mathcal{V} = \mathcal{V}$ $ \mathcal{V} \neq \mathcal{V}$ $\mathcal{V} := \text{Event.name}$ $ \text{Event.sender}$ $ \text{Event.receiver}$ $ \text{Event}.\langle payload_field \rangle$ $ \langle const \rangle$ (a) Scenario Language.	$\text{S1}(e_0, e_1) :$ $e_0 \prec e_1 \wedge e_1.\text{key} = e_0.\text{key} \wedge$ $e_0.\text{name} = \text{"writeReq"} \wedge$ $e_1.\text{name} = \text{"readReq"} \wedge$ $\text{S2}(e_0, e_1, e_2, e_3) :$ $e_0 \prec e_1 \prec e_2 \prec e_3 \wedge$ $e_1.\text{name} = \text{"writeReq"} \wedge$ $e_3.\text{name} = \text{"readReq"} \wedge$ $e_3.\text{key} = e_1.\text{key} \wedge$ $e_0.\text{name} = \text{"newLeaderElected"} \wedge$ $e_2.\text{name} = \text{"newLeaderElected"} \wedge$ $e_0.\text{leader} \neq e_2.\text{leader}$ (b) Examples.
--	--

Figure 4.4: Scenario specification language and example scenarios for the key-value store in § 4.2.3

4.3.5 Scenario Specification Language

Drawing inspiration from complex event processing tools like FlinkCEP [9] and Esper [25], we design a scenario specification language to guide FEST toward exploring targeted behaviors.

Figure 4.4a presents the simplified grammar for the scenario specification language. Each scenario ($\mathcal{S}$) has a name $\langle name \rangle$, a list of events, and a set of constraints over these events. The language supports two types of constraints ($\mathcal{C}$): (1) *happens-before constraint* ($\text{Event} \prec \text{Event}$) that capture the expected temporal order of these events; (2) *equality constraints* like values equal ($\mathcal{V} = \mathcal{V}$) and values unequal ($\mathcal{V} \neq \mathcal{V}$) that are used for adding constraints over the contents of the events. Every event has the following fields over which constraints can be defined: name (name of the event), sender (sender of the event), receiver (receiver of the event), and the payload associated with the event referred to as *payload_field*.

Figure 4.4b shows the scenarios S1 and S2 described in § 4.2.3 specified in our scenario language. S1 scenario is defined over two events `writeReq` and `readReq` where we would like to ensure that a write (e_0) is followed by a read (e_1), both operations performed on the same key. Scenario S2 describes a sequence of four events, a write (e_1) and a read (e_3) with the same key occurring before and after an election event (e_2) and processed by different leaders ($e_0.\text{leader} \neq e_2.\text{leader}$).

Now that we can define custom scenarios, we say that an execution having event sequence E *satisfies* a scenario $Scen$ which is parameterized on events $e_1, \dots, e_n$ if and only if there exists a (not necessarily contiguous) subsequence of events $e_1, \dots, e_n$ in E such that $Scen(e_1, \dots, e_n)$ is true.

4.3.6 Compliance Feedback

Given a scenario, the purpose of the compliance measurement is to determine to what extent the user-provided scenarios are satisfied by generated schedules. Ideally, we want to find schedules that satisfy all constraints in the scenario. However, finding schedules that partially satisfy the scenario may indicate that the schedule is close to our objective. So, the compliance measurement calculates the fraction of constraints satisfied in a scenario.

For example, given scenario S1 (ref. Fig. 4.4b), if the execution results in an event sequence `[..., writeReq("key1", ...), ..., readReq("key2", ...), ...]`, then FEST identifies that the first three constraints are satisfied—there was a *write* before a *read*. However, the fourth constraint, stipulating that the write and read requests should have the same *key*, is not satisfied. So, the scenario compliance in this case is $\frac{3}{4}$.

4.4 Evaluation

Our evaluation addresses the following research questions by comparing FEST with state-of-the-art testing techniques:

- (RQ1) How does FEST perform when the goal is to improve behavioral coverage?
- (RQ2) How does FEST perform as a bug-finding technique?
- (RQ3) How does FEST perform when testing specific scenarios?
- (RQ4) Does FEST priority-based schedule generation help FEST in covering more timelines?

4.4.1 Implementation

We implemented FEST on top of the open-source P framework [18]. FEST extends P in three important ways: (1) extending the existing P language to add support for specifying scenarios, (2) extending the existing P checker to collect feedback dynamically, and (3) a feedback-guided adaptive schedule generator for the P checker. FEST is implemented as an external guiding tool that observes, generates schedules, and forwards them to the system orchestrator (P Checker in our case). Our implementation has been merged into the official P repository and is now used by the entire P community. Note that the ideas implemented in FEST can be applied to test any other distributed system that supports an orchestrator to control schedules and collect feedback.

4.4.2 Comparison and Benchmarks

In our evaluation, we compare the feedback-guided adaptive scheduling approach in FEST against four state-of-the-art testing approaches for distributed systems: PCT [16, 17], POS [78], POS with conflict analysis (POS+) [77] and traditional RL based schedule generation (QL) [17, 52]. FEST implements three feedback-guided adaptive schedule generation algorithms, which are referred to as FEST_{PCT} , FEST_{POS} , and $\text{FEST}_{\text{POS+}}$. For PCT and FEST_{PCT} , we use the bound of $\text{depth} = 3, 15, 50$, which performed best for our benchmarks. As PCT and POS were originally designed for concurrency testing, we adapt these algorithms for distributed systems as proposed

Table 4.1: Description of case studies from AWS and open-source models. ‘LOC’ indicates the P line of code of the formal model of the system.

AWS System	LOC	AWS System	LOC
Elastic Storage	4539	CDN	1291
Leader Election	2748	Bucket Storage	994
Stream Log	1495	Financial Service	4095
Memory Database	3219	Message Broker	1456
Network Dynamic Scaling	4295	Open-source System	LOC
Network Routing	5392	German [62]	282
Database Storage Engine	3955	OSR [62]	381
Access Management	3298	Paxos [62]	598
Concurrency Control	3056	2PhaseCommit [18]	822
FreeRTOS	1485	DeepSeek-3FS [4]	4648

in prior works [17, 77]. For RQ4, we implemented a baseline approach (NoPrio) that performs feedback-guided adaptive scheduling without priority-based scheduling by assigning a uniform diversity value of 1 to all saved recordings (as referenced in Algorithm 5, Line 26).

Comparison metric. We use abstract Lamport timeline [39, 51] to represent abstract system behavior explored. If a scenario is provided, we measure the number of unique timelines explored that satisfy the input scenario. We run each technique for 1 hour for each benchmark and repeat the experiment 5 times to account for randomness. All experiments were performed on a cloud instance with an AMD EPIC processor with 64 cores and 256 GB memory.

Benchmarks. To evaluate behavior coverage and bug-finding capability in real-world systems, we applied FEST to 19 P models of distributed services, as shown in Table 4.1, covering 94 test cases. Each test case includes different system configurations, initial states, and assertions. Four models, with 13 test cases, are from prior research [18, 21, 62], while 14 protocol models from AWS span databases, storage, networking, and routing, accounting for 69 test cases. Examples include a control plane protocol for merging and splitting partitions in distributed storage, a leader election protocol, and a conflict resolution protocol for an in-memory database. We also included the latest released DeepSeek 3FS model [4], which contains 12 test cases. The benchmark includes 27 known bugs, and the entire evaluation took over 200 CPU days, making it the largest evaluation of distributed system model testing to date.

To evaluate the coverage of the scenario, we collaborated with developers and implemented 7 scenarios derived from their testing case studies shown in Table 4.2 and applied these scenarios to 4 models. We present the aggregated results in the following sections.

4.4.3 RQ1: Timeline Coverage Improvement

Figure 4.5 shows the timeline coverage improvements using FEST. The x-axis represents the improvement factor on a logarithmic scale, while the y-axis shows the proportion of instances achieving at least that level of improvement. The results demonstrate that the feedback-guided schedule generation approach (FEST) improves the existing techniques. Specifically, FEST enhances timeline coverage for PCT3 in 95% of test cases, with an average improvement of 86%. Similar improvements are observed for PCT15 and PCT50, with average timeline coverage

Table 4.2: Description of testing scenarios.

Name	AWS Target	Description
SC	Elastic Storage	Perform concurrent splits and merge twice on the same partition.
SR	Leader Election	A stable leader processes read requests.
CR	Leader Election	A new leader is elected while an old leader is servicing read requests.
RD	Stream Log	Stream log replicator handles read requests.
RT	Stream Log	Stream log replicator handles a rotation requests while servicing reads.
SL	Stream Log	Stream log replicator handles a replica partition requests while servicing reads.
CI	Memory Database	During a node failure, two requests set and increase the same key concurrently.

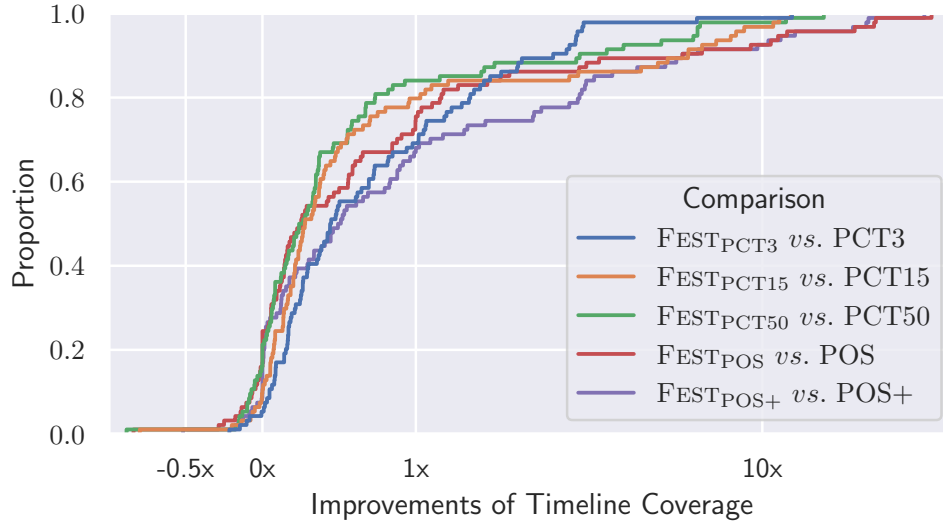


Figure 4.5: Cumulative distribution function of improvements over the original approach without feedback (e.g., $\text{FEST}_{\text{PCT3}}$ versus PCT3) for different schedule generation algorithms.

improvements of 133% and 95%, respectively. For POS, FEST achieves a $2.2\times$ improvement, and for POS+, a $2.4\times$ improvement, with maximum improvements reaching $41\times$ and $39\times$, respectively. FEST has improved or achieved the same timeline coverage for 88% of cases. On average, FEST achieves a $1.53\times$ improvement in timeline coverage, with a maximum improvement of up to $41\times$ as compared to the original technique.

There are a few cases (12%) where FEST achieves lower timeline coverage, with 75% of the reductions being less than 4%. The reason for this was that the number of timelines explored is roughly proportional to the number of schedules generated for these models. Since the feedback algorithm introduces a small overhead, FEST generates fewer schedules per unit of time, leading to lower timeline coverage.

Achieve Highest Timeline Coverage. Figure 4.6 illustrates the number of test cases for which each technique achieves the highest timeline coverage. The FEST_{PCT} technique with a bound

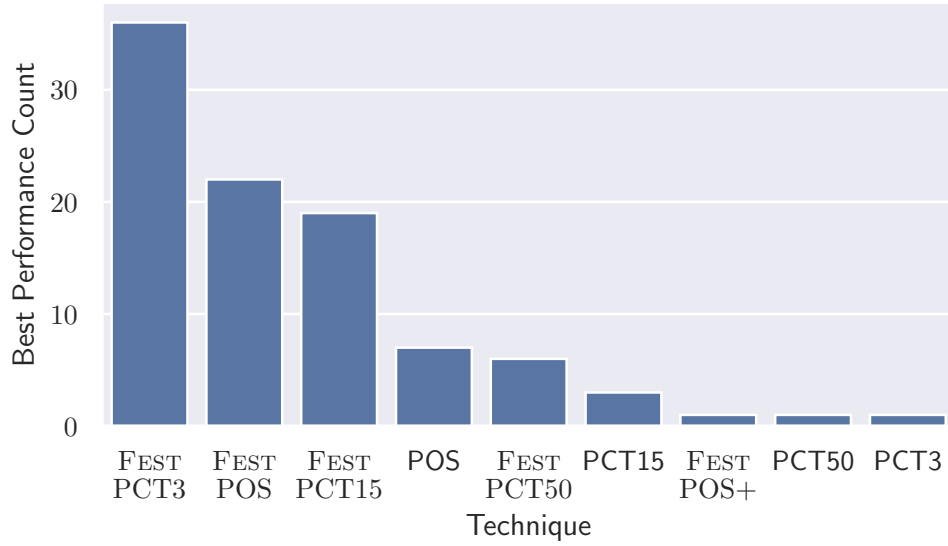


Figure 4.6: Best performance count: Number of test cases for which each technique achieves the highest timeline coverage.

of 3 performs best, achieving the highest timeline coverage in 36 test cases. The FEST_{POS} technique and FEST_{PCT} technique with a bound of 15 achieve the highest timeline coverage in 22 and 19 test cases, respectively. Additionally, the FEST_{PCT} technique with a bound of 50 achieves the highest timeline coverage in 6 test cases. Overall, one of the FEST-based approaches achieves the highest timeline coverage in 88% of the test cases. Our result also shows that QL is ineffective in exploring diverse timelines and does not achieve the highest timeline coverage in any test case. QL imposes no scheduling constraints, creating an exponentially large search space that its feedback mechanism cannot navigate efficiently. This highlights the importance of feedback-guided adaptive design with a bounded search space.

Answer to RQ1: FEST improves existing RSBT algorithms to achieve higher timeline coverage.

4.4.4 RQ2: Bug Finding Capabilities

Figure 4.7 displays the results using an upset plot, a type of visualization that showcases the intersections of sets, highlighting the bug-detection capabilities of each technique for rediscovering the known bugs in our benchmark. Overall, $\text{FEST}_{\text{PCT3}}$ and $\text{FEST}_{\text{PCT15}}$ proved superior, identifying 16 bugs, while PCT3, PCT15, PCT50, POS, and POS+ identified only 10 and QL only identified 12. Notably, $\text{FEST}_{\text{PCT3}}$ and $\text{FEST}_{\text{PCT15}}$ capture all bugs detected by PCT, POS, and QL and exclusively discovered an additional 2 bugs missed by the other algorithms. This demonstrates that exploring diverse schedules helps discover more bugs.

Answer to RQ2: FEST demonstrates superior bug detection capabilities in distributed system testing by identifying the highest number of unique bugs.

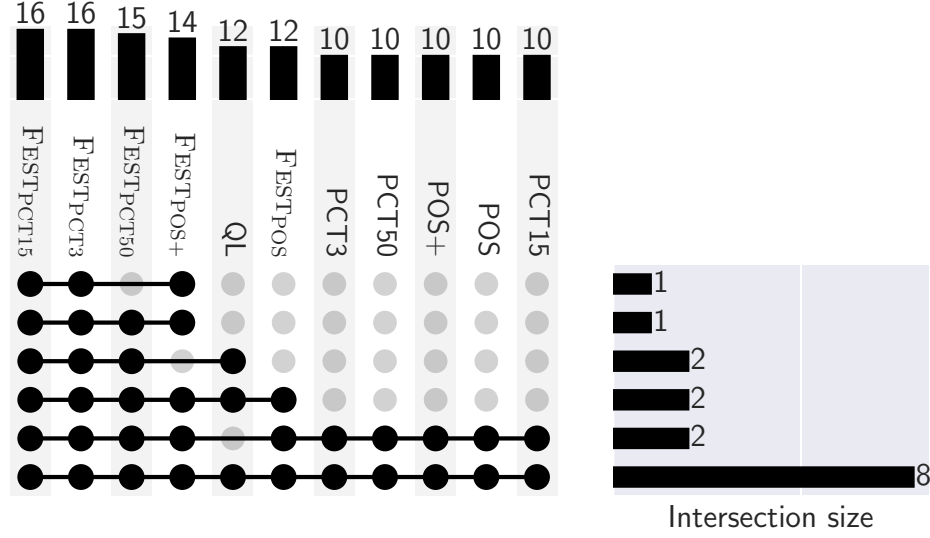


Figure 4.7: Upset plot for bug counts across different scheduling algorithms. Higher is better.

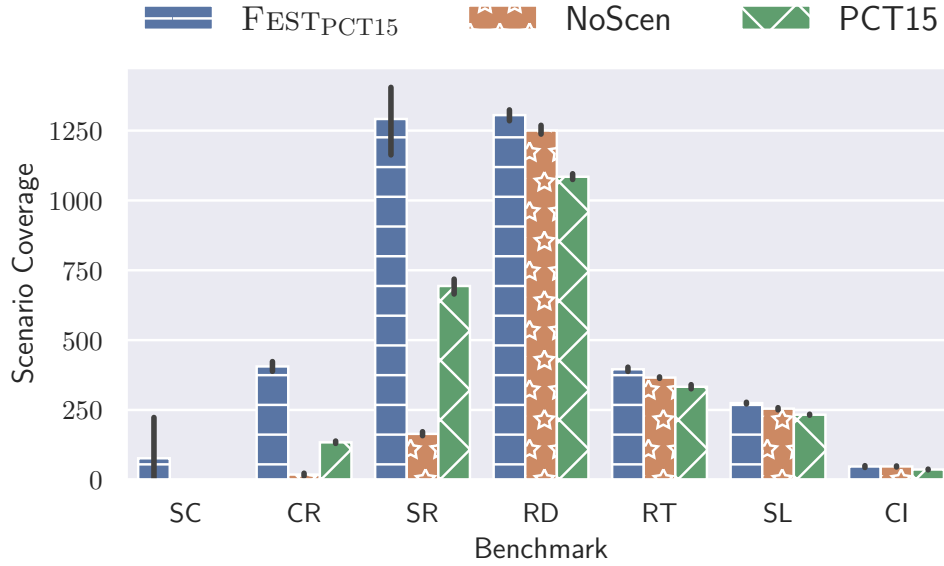


Figure 4.8: Comparison of timeline coverage that satisfies the specified scenario across different techniques.

4.4.5 RQ3: Scenario Coverage

To evaluate the contribution of scenario-guided schedule generation, we compare FEST with scenarios against FEST implementation that ignores the given scenario feedback (NoScen). Both FEST and NoScen uses the diversity feedback. We use the input scenario described in Table 4.2.

Figure 4.8 shows the scenario coverage achieved by different techniques. To save space, we present results from PCT15 and FEST_{PCT}. Across all benchmarks, FEST_{PCT} either outperforms

or matches NoScen, highlighting the value of incorporating scenario feedback into feedback guidance. Moreover, $FEST_{PCT}$ consistently outperforms PCT on all benchmarks. In the elastic storage model with the concurrent split-merge (SC) scenario, $FEST_{PCT}$ identifies up to $39\times$ more unique timelines than both PCT and NoScen. For leader election protocol with concurrent read (SR) and stable read (CR) scenarios, FEST shows $2.7\times$ and $1.7\times$ improvements compared to PCT, respectively. For stream log scenarios (RD, RT, and SL) and memory DB with transition read scenario (CI), the improvement $FEST_{PCT}$ is between 10% to 20%. This diminished advantage stems from the inherent simplicity of the two systems (Stream Log and Memory Database), leading to a smaller state space to explore.

Our findings notably reveal that, in the absence of scenario coverage, developers utilizing the P checker with the PCT technique remain unaware of significant coverage gaps in various scenarios, such as Elastic Storage with the SC scenario. This underlines the critical need to incorporate scenario coverage in the testing process. This feedback is essential for developers to accurately gauge and enhance the thoroughness of the tests conducted on their systems.

Scenarios helped find new bugs: FEST also helped uncover new bugs in AWS Elastic Storage design that other techniques failed to find. In the Elastic Storage model, it failed to generate enough requests to trigger split-merge operations, resulting in low timeline coverage in the SC scenario. Once this bug was fixed, FEST immediately identified another bug related to how the model handles failures during the split-merge operation. Hence, FEST helped eliminate a critical bug during the design phase, which was missed by other search techniques in P checker. These findings emphasize the pivotal role of scenario-based feedback, helping developers ensure that critical scenarios are explored during design validation and that the test generators can trigger these scenarios.

Answer to RQ3: The scenario feedback significantly enhances scenario coverage, especially in complex models. The scenario coverage also helps developers gain feedback about scenarios that are inadequately tested.

4.4.6 RQ4: Priority-based Schedule Generation

Figure 4.9 illustrates the timeline coverage improvement of $FEST_{PCT15}$ compared to the approach without priority-based scheduling (NoPrio). The x-axis shows the benchmarks, while the y-axis represents the improvement in timeline coverage. Across all benchmarks, the priority-based scheduling approach achieves better coverage for 84% benchmarks, with an average improvement factor of $1.22\times$. This demonstrates that incorporating priority into the schedule generation enables FEST to explore more diverse timelines more effectively.

Answer to RQ4: Priority-based schedule generation enhances the efficiency of FEST in covering more unique timelines.

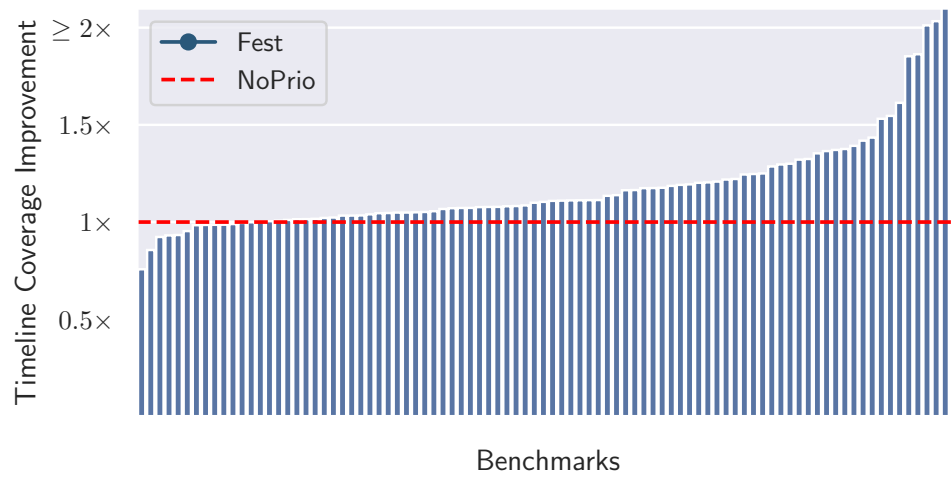


Figure 4.9: Timeline coverage improvement of $\text{FEST}_{\text{PCT15}}$ compared to the version without priority-based scheduling (NoPrio). The y-axis is cut off at $2\times$ due to space limitations.

Chapter 5

Diorama: Microcosm Distributed System Testing

Testing distributed systems requires reasoning about three tightly coupled sources of nondeterminism: thread interleavings within a node, message interleavings across nodes, and time-dependent behavior such as delays, timeouts, and timer-triggered events. Real failures often arise only from specific combinations in this joint space.

Prior work exposes a persistent tradeoff between control and efficiency. Techniques such as FlyMC [50], SAMC [41], and Mallory [51] intercept network-facing libraries and primarily explore message nondeterminism. These systems are efficient, but they largely ignore thread-level concurrency and offer limited control over time. MODIST [74] moves lower in the stack by introducing a system proxy that can influence both network and thread behavior, but it still lacks precise control over time. At the other extreme, systems such as Antithesis use a deterministic hypervisor to control the application together with the operating system, achieving strong control at substantial cost. Lower-layer approaches are also a poor fit for distributed systems implemented in managed languages such as Java and the CLR. Because proxies and hypervisors operate below the application layer, they primarily control the language runtime rather than the application itself, which limits visibility into application-level synchronization and semantics and wastes effort exploring behaviors that are irrelevant to the target program logic.

Testing distributed systems is ultimately about validating *application behavior*, not revalidating the network stack, operating system, or language runtime. For testing purposes, faithfully executing the concrete network layer or wall-clock time is unnecessary; what matters is preserving the *observable effects* of message delivery and time progression on the application.

This chapter builds on a simple observation: from the application’s perspective, the effects of network communication and time can be modeled through abstractions implemented using shared memory and concurrency primitives. Message passing can be modeled as reads and writes to message queues, and time can be modeled through accesses to a monotonic counter. This view lets us reduce a distributed execution to a single-process, multithreaded execution without losing the behaviors relevant to testing. The result is a unified search space in which network nondeterminism, node-local concurrency, and timing behavior all become scheduler-controlled choices.

Based on this insight, we present DIORAMA, a *microcosm framework* that reduces distributed-

system testing to single-process concurrency testing. DIORAMA replaces the network layer with in-memory message queues and replaces real time with a shared counter. By turning message delivery and time advancement into schedule choices, DIORAMA enables systematic exploration of executions that involve both inter-node and intra-node concurrency while avoiding the overhead and opacity of lower-layer control.

5.1 Background

In this section, we present a simplified example of a latent concurrency bug in AWS Route53’s DNS management system, as described in a recent incident report [8].

5.1.1 The AWS Route53 Incident

On October 19, 2025, Amazon Web Services experienced a significant outage affecting Route53, their authoritative DNS service [8]. The incident began when a routine software deployment triggered a race condition. The bug caused widespread unavailability of DNS resolution services, which in turn disrupted numerous applications and services that depend on Route53 for domain name resolution.

The root cause involved a complex interaction between thread-level concurrency within individual Route53 nodes and message-level concurrency across the distributed system. As described in the incident report, Route53 nodes maintain distributed state about DNS zone configurations using a consensus-based replication protocol. Each node runs multiple worker threads that process incoming DNS queries while simultaneously participating in replication to maintain consistency across the cluster.

The specific bug manifested when the following conditions occurred simultaneously:

1. A configuration update message arrived at a node, triggering a local state transition.
2. Multiple worker threads on the same node concurrently accessed the partially updated configuration.
3. Additional replication messages from peer nodes arrived before the local update completed.

Figure 5.1 illustrates this scenario. The system consists of two types of nodes: *Route 53* nodes that handle DNS queries and manage DNS records, and *Enactor* nodes that process and validate DNS record updates. The Route 53 node contains multiple worker threads that handle incoming DNS queries. In this example, Worker 1 sends a `Register(v1)` message to Enactor 2, while Worker 2 sends a `Register(v2)` message to Enactor 1. The Enactor nodes process these DNS record updates and send back `Apply` messages: Enactor 1 responds with `Apply(v1)` and Enactor 2 responds with `Apply(v2)`. These apply messages arrive back at the Route 53 node, where the worker threads process them concurrently.

The race condition manifests through the interleaving of operations across the two worker threads as they process the apply messages. Worker 1 receives `Apply(v2)` and begins executing a series of operations: `Fetch Store` to retrieve and store the new configuration data, `Update Endpoints` to modify the active endpoints, `Update Version` to record the new version, and finally `Cleanup(v2)` to remove data from the old version v1. Meanwhile, Worker 2

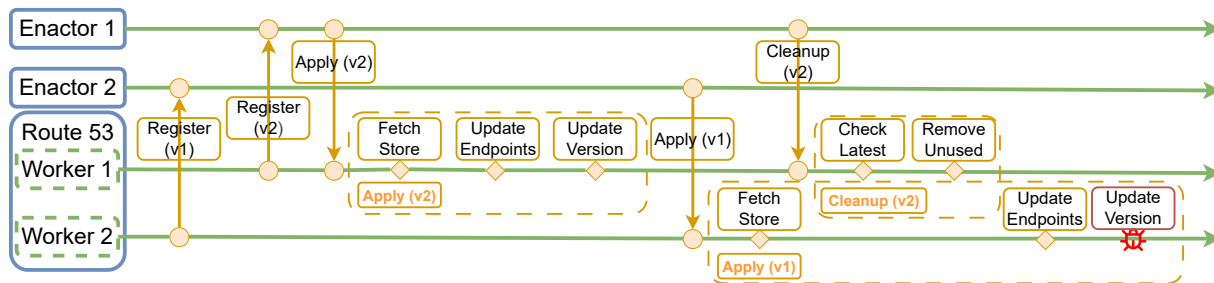


Figure 5.1: An illustration of the AWS Route53 race condition caused by a complex interaction between network messages and thread interleavings. Two Enactor nodes send different configuration versions (v1 and v2) to the Route 53 node. Worker threads within Route 53 process these updates concurrently, leading to a race condition where operations on different versions interleave incorrectly. The red bug icon marks the point where the inconsistency is detected during a version update.

receives `Apply (v1)` and begins its own sequence of operations. The critical bug occurs when Worker 1 executes `Cleanup (v2)`—which removes data associated with version v1—while Worker 2 is still in the middle of processing `Apply (v1)`. This creates a scenario where Worker 2’s subsequent operations (`Fetch Store`, `Check Latest`, `Remove Unused`, `Update Endpoints`, `Update Version`) operate on partially cleaned-up state, ultimately leading to an inconsistent version update (marked with the red bug icon).

Despite the fact that this is a well-studied atomicity violation [46], finding this bug is extremely challenging in practice. Reproducing the failure requires setting up multiple machines (the Route 53 node and at least two Enactor nodes), coordinating DNS record registrations that generate conflicting version updates, and then precisely controlling both the message interleaving (the order in which `Apply` messages arrive from the Enactors) and the thread interleaving (the execution order of operations within the worker threads) to hit the narrow window where `Cleanup (v2)` executes between Worker 2’s operations on v1.

5.1.2 Timed Operations

Beyond message passing and thread concurrency, time introduces a third critical dimension of nondeterminism in distributed systems. As noted by Waldo et al. [72], distributed systems must explicitly account for latency, and this awareness of time manifests throughout real-world distributed systems in ways that profoundly affect their behavior and correctness.

Time-dependent operations appear pervasively in distributed systems for three primary reasons: *coordination*, *resource management*, and *fault detection*. Unlike concurrent programs running on a single machine, where a thread can inspect shared memory or query the state of another thread directly, nodes in a distributed system have no way to observe one another’s state except by exchanging messages over an unreliable network. This fundamental limitation means that every question a node asks about the outside world—“Is my peer alive?”, “Has the leader changed?”, “Did my write commit?”—must be answered indirectly, through messages that may be delayed, reordered, or lost. Time-based mechanisms such as heartbeats, timeouts, and periodic polling

exist precisely to bridge this gap: they let a node make probabilistic inferences about remote state in the absence of direct observation.

Heartbeats and health monitoring. Heartbeat mechanisms are among the most ubiquitous time-dependent patterns in distributed systems [11]. A heartbeat is a periodic signal sent from one component to another to indicate liveness. The sender commits to broadcasting its heartbeat at a regular interval, and the receiver monitors these signals: if no heartbeat arrives within a configured timeout, the receiver declares the sender failed. The choice of heartbeat interval and timeout involves a fundamental trade-off—shorter intervals enable rapid failure detection but generate substantial network overhead, while longer intervals conserve resources at the cost of slower detection [11]. A typical rule of thumb is to set the failure timeout to at least two to three times the heartbeat interval, and to require multiple missed heartbeats before declaring a node dead, reducing false positives from transient packet loss or momentary delays.

Real-world systems implement heartbeats in a variety of ways. In the *push model*, the monitored node actively sends heartbeat messages at regular intervals, as used by Kubernetes kubelets (which send node status updates every 10 seconds), Hadoop YARN NodeManagers, and Celery workers. In the *pull model*, a monitoring system periodically polls nodes for liveness, as used by load balancers probing backend health endpoints. Many production systems combine both approaches [11].

```
1 while (true) {
2     Request request = new Request("Heartbeat");
3     request.send(leader);
4     Thread.sleep(500); // wait 500ms before next heartbeat
5 }
```

As distributed systems grow in scale, centralized heartbeat monitoring becomes a bottleneck and a single point of failure. To address this, systems such as Apache Cassandra use *gossip protocols* for failure detection [11, 70]: each node periodically exchanges membership information with a small number of randomly selected peers, and failure information propagates through the cluster in a decentralized fashion.

Timeouts and failure detection. More generally, distributed systems use timeouts to distinguish between slow responses and failed nodes. In the absence of perfect failure detectors, timeouts serve as the primary mechanism for detecting and recovering from partial failures. A request with a timeout encodes an assumption about the maximum acceptable response time:

```
1 Response response = request.send(replica, timeout);
2 if (response == null) {
3     // timeout expired: treat replica as failed
4     failover(replica);
5 }
```

Fixed timeouts are simple but brittle: a single threshold cannot account for variable network latency. More sophisticated approaches adapt over time. The *Phi Accrual Failure Detector* [11, 30] replaces the binary alive-or-dead decision with a continuous suspicion level derived from the statistical distribution of recent heartbeat inter-arrival times, allowing the same configuration to work across diverse network environments without manual tuning. Similarly, *exponential*

backoff progressively increases the interval between retries after successive failures, avoiding overwhelming an already-struggling node.

The interplay between timeouts and actual network delays creates a rich space of timing-dependent behaviors. A system that works correctly when all operations complete within their timeout windows may fail catastrophically when network delays cause timeouts to expire, triggering cascading failovers or split-brain scenarios.

Rate limiting and polling. Time-dependent operations also regulate resource usage and implement polling-based protocols. Beyond heartbeats, systems use polling loops with timeouts to wait for conditions to become true:

```
1 while (true) {
2     Request request = new Request("IsJobDone?");
3     Response response = request.send(worker, 1000); // 1s timeout
4     if (response.getContent().equals("Done")) {
5         break;
6     }
7 }
```

These patterns appear extensively in distributed consensus protocols (e.g., Raft, Paxos), distributed task schedulers, and coordination services. The chosen timeout values directly affect system behavior: shorter timeouts reduce latency but increase the risk of false positives (declaring a node dead when it is merely slow), while longer timeouts improve robustness at the cost of slower failure detection.

Testing challenge. These time-dependent behaviors pose significant challenges for testing. Real-time testing can only explore timing behaviors that happen to occur during test execution, and deliberately injecting delays requires a thorough understanding of how time is used in the system under test. Naively delaying or dropping messages can make testing ineffective: for example, if a testing tool injects too many delays into heartbeat messages, every node in the cluster will conclude that all of its peers have failed, causing the system to stop making progress entirely rather than exercising the interesting application logic that the test intends to explore. Conversely, injecting too few delays may never trigger the timeout-driven recovery paths where bugs hide.

5.2 Design

The central idea behind DIORAMA is to reduce a distributed execution into a single-process, multithreaded execution that preserves all application-observable behaviors. As shown in Figure 5.2, DIORAMA replaces the real network with in-memory message queues, replaces wall-clock time with a scheduler-controlled logical clock, and isolates nodes from one another using per-node class loaders—all within a single JVM process. The result is a *microcosm*: a miniature, fully controlled replica of the distributed system in which every source of nondeterminism—thread interleavings, message orderings, and time progression—becomes a scheduling decision that can be explored systematically.

The remainder of this section describes the three key mechanisms in detail: network virtualization (§5.2.1), time virtualization (§5.2.2), and node isolation via class loading (§5.2.3).

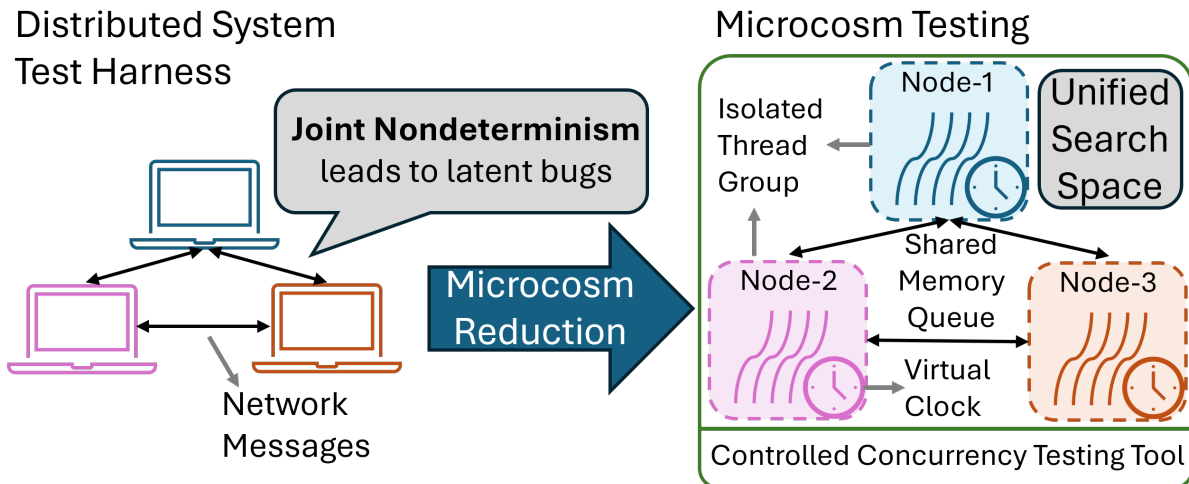


Figure 5.2: Architecture of DIORAMA. Each logical node runs in an isolated class-loader within a single JVM. The network layer is replaced by in-memory message queues, and wall-clock time is replaced by a logical clock. A unified scheduler controls thread interleavings, message delivery, and time advancement.

5.2.1 Network Virtualization

Distributed nodes communicate by sending and receiving messages over TCP sockets. Rather than routing traffic through the operating system’s network stack, DIORAMA intercepts socket operations at the Java API boundary and redirects them to in-memory data structures that faithfully emulate the socket semantics visible to the application.

DIORAMA uses bytecode instrumentation to intercept constructor calls and method invocations on the `java.net.Socket` and `java.net.ServerSocket` classes. When application code creates a new `Socket` or `ServerSocket`, the instrumentation redirects the call to a corresponding *mocked* implementation: `MockedSocket` or `MockedServerSocket`. These mocked classes implement the same state-machine semantics as their real counterparts—tracking bound, connected, listening, and closed states—but all data transfer occurs through in-memory buffers rather than kernel sockets.

A `MockedServerSocket` maintains a pending-connection queue. When a server node calls `accept()`, the server thread blocks on this queue until a connection arrives. On the client side, calling `connect()` on a `MockedSocket` with a target address enqueues a connection request into the corresponding server’s pending-connection queue and blocks until the server accepts. A global address registry maps logical addresses (host–port pairs) to `MockedServerSocket` instances, enabling nodes to discover one another without relying on real DNS or network interfaces.

Once a connection is established, the two endpoints are linked by a pair of in-memory streams—one for each direction. Writes to the output stream of one endpoint append data to the input stream of the other. To notify the receiver, each write operation wakes up any selector or blocking read that the receiving socket has registered. Because both endpoints reside in the same JVM, the data transfer is a simple memory copy with no serialization overhead beyond what the application

```

1  volatile long logicalTime = 0;
2  public void startMachine() {
3      // ...
4      new Thread(() -> {
5          while (machineIsRunning) {
6              logicalTime++;
7          }
8      }).start();
9      // ...
10 }

```

Figure 5.3: Per-machine time thread. Each logical node maintains its own logical clock, incremented by a dedicated background thread whose scheduling is controlled by DIORAMA.

```

1  public void sleep(long millis) {
2      long deadline = logicalTime + millis;
3      while (logicalTime < deadline) {
4          Thread.yield(); // yield to scheduler
5      }
6  }

```

Figure 5.4: Virtualized sleep. DIORAMA replaces `Thread.sleep()` with a yield loop that waits for the logical clock to reach the target deadline.

itself performs.

From the scheduler’s perspective, each blocking socket operation—`accept()`, `connect()`, `read()`—is a scheduling point. The scheduler decides which blocked thread to resume and when, giving it direct control over message delivery order. Dropping a message or delaying a delivery is equivalent to not scheduling the corresponding read. This turns network-level fault injection into a scheduling policy, requiring no separate fault-injection infrastructure.

5.2.2 Time Virtualization

Testing distributed systems under different timing conditions is important but challenging. DIORAMA addresses this by replacing wall-clock time with a logical clock on each machine. Through bytecode instrumentation, DIORAMA intercepts calls to `System.currentTimeMillis()` and `System.nanoTime()` across all loaded application classes, and each intercepted call returns the current value of a *per-machine* logical counter rather than the real system time. Because advancing the logical clock is itself a scheduling decision, DIORAMA can systematically explore different timing variations while preserving the causal ordering of time-dependent events.

As shown in Figure 5.3, when a machine starts, DIORAMA spawns a background thread that continuously increments the logical clock. Because the scheduler decides when this thread runs, the rate at which time advances is under the scheduler’s control. The scheduler can let the time thread run many steps before scheduling an application thread, simulating a slow node, or it can delay the time thread to simulate fast execution relative to wall-clock expectations.

For timed-blocking operations such as `Thread.sleep()`, DIORAMA replaces the call with

a loop that yields to the scheduler until the logical clock reaches the desired deadline (Figure 5.4). This design gives the scheduler direct control over how other threads interleave with the sleeping thread’s wake-up. For example, the scheduler can choose to advance time past a timeout *before* a pending message is delivered, or it can deliver the message first—systematically exploring the boundary between timely and late responses, which is precisely where many timing-related bugs hide.

Fast-forwarding is also possible: when the scheduler determines that no application threads are runnable until a future logical time, it can advance the time thread directly to that point, skipping idle periods without wasting CPU cycles on busy-waiting.

5.2.3 Node Isolation via Class Loading

Running multiple distributed nodes inside a single JVM creates a fundamental isolation challenge: nodes that share the same loaded classes will inadvertently share static fields, class-level state, and singleton instances, leading to interference that would never occur in a real deployment where each node runs in a separate process. DIORAMA solves this problem by leveraging the JVM’s class-loading mechanism to create isolated namespaces for each logical node.

For each logical node in the microcosm, DIORAMA creates a dedicated `DioramaClassLoader`. Each class loader loads application classes from the node’s configured classpath independently of the other nodes’ class loaders. Even though two nodes may run the same application code, each class loader produces a distinct `Class` object for every class it loads. Because static fields are scoped to a `Class` object, and `Class` objects are scoped to their defining class loader, each node gets its own private copy of all static state.

5.2.4 Lifecycle

A practical microcosm must manage the full lifecycle of each logical node: startup, workload execution, and teardown.

Each logical node is wrapped in a `ServerManager` that exposes two operations: `run`, which starts the node with its command-line arguments, and `stop`, which terminates the node and releases its resources. DIORAMA does not distinguish between “servers” and “clients”; a client is simply a node whose `run` method connects to other nodes, issues requests, and exits, potentially leaving its `stop` method empty.

The test harness prepares the environment for each node—creating configuration files, setting system properties, and specifying classpaths—then launches each node through DIORAMA’s `launchServer` API. Each launch allocates a new class loader, spawns the node’s main thread within that class loader’s scope, and registers the node with the scheduler. The scheduler begins exploring interleavings as soon as the first scheduling point is reached.

When the test harness signals completion, DIORAMA invokes each node’s `stop` method in reverse order of startup. Background threads spawned by the application (e.g., heartbeat threads, metrics collectors) may resist graceful shutdown. In such cases, DIORAMA falls back to forcibly terminating all threads associated with the node’s class loader by injecting exceptions, ensuring that no residual threads leak across test iterations.

5.3 Case Study

We now demonstrate how DIORAMA can be used to find the critical race condition that caused the AWS Route53 outage described earlier in this chapter. We implement simplified versions of the three services involved in the incident—Route53, the DNS Enactor, and the DNS Planner—and run them inside a single DIORAMA microcosm. We then compare DIORAMA’s ability to discover the failure-inducing interleaving against `rr` [56], a record-and-replay tool that captures nondeterministic thread schedules by recording actual executions.

5.3.1 Implementation

Our case study implementation consists of three services that mirror the architecture described in the AWS incident report [8].

Route53 Simulator. The Route53 simulator maintains an in-memory plan store (a `ConcurrentHashMap` mapping version numbers to DNS plans), an endpoint-binding table that maps DNS endpoint names to the version of the plan currently serving them, and a monotonically increasing version counter. It exposes four operations over a socket-based JSON protocol:

- `registerPlan`: stores a new DNS plan and assigns it the next version number;
- `applyPlan`: given a version, looks up the corresponding plan, updates the endpoint bindings, and records the version as the last applied;
- `deleteOlderThan`: removes all plans whose version is strictly less than a given threshold and not equal to the last applied version;
- `resolve`: returns the DNS records currently bound to a given endpoint.

Crucially, these operations use `ConcurrentHashMap` and `AtomicLong` for individual field accesses but do *not* provide atomicity across the multi-step `applyPlan` sequence (fetch plan, update endpoints, update version). This faithfully models the race condition described in the incident report.

DNS Enactor. Each DNS Enactor instance receives a plan name from the Planner, fetches the full plan, and then executes a three-step sequence against Route53: (1) register the plan to obtain a version number, (2) apply the plan, and (3) delete plans older than the newly applied version. Two independent Enactor instances run concurrently, modeling the redundant, multi-AZ deployment described in the incident report.

DNS Planner. The Planner creates DNS plans specifying endpoint-to-record mappings and distributes them to Enactors in round-robin fashion. In the test scenario, the Planner first publishes a baseline plan and waits until it is visible via Route53 resolution. It then concurrently publishes two additional plans (`plan1` and `plan2`), which are routed to different Enactors, setting up the conditions for the race.

Table 5.1: Happens-before pair exploration across 1,000 iterations. DIORAMA explores a strict superset of the orderings reached by `rr`, covering 47 additional pairs.

Category	# pairs	% of 600
Both	385	64.2%
DIORAMA only	47	7.8%
<code>rr</code> only	0	0.0%
Neither	168	28.0%

Test oracle. After all plans have been published and sufficient time has elapsed for the Enactors to process them, the test oracle checks whether every DNS endpoint can still be resolved. A failure occurs when `resolve` returns an empty result for any endpoint—indicating that the race condition has caused DNS records to be deleted, reproducing the outage.

5.3.2 Test Setup

The microcosm launches four logical nodes within a single JVM: one Route53 simulator, two DNS Enactors, and one DNS Planner. Each node runs in its own isolated class loader and communicates through DIORAMA’s in-memory network layer. The DIORAMA scheduler controls all thread interleavings, message delivery order, and time progression across the four nodes.

For comparison, we also run the same simulation under `rr` [56]. We use `rr`’s chaos mode, which perturbs the thread schedule by injecting random delays but it has no control over message interleaving or time progression, both of which remain governed by the operating system’s default behavior.

Both tools execute the simulation for 1,000 iterations. Each iteration starts from the same initial state, publishes the baseline plan, and then concurrently publishes two conflicting plans through different Enactors.

5.3.3 Results

Bug detection. DIORAMA successfully identifies the failure within 1,000 iterations: at least one iteration produces a schedule in which `resolve` returns empty for a DNS endpoint, reproducing the outage scenario. In contrast, `rr` fails to trigger the bug in any of its 1,000 iterations. This is because the failure requires a precise interleaving in which one Enactor’s cleanup operation executes between another Enactor’s apply and version-update steps.

Schedule diversity. To understand *why* DIORAMA succeeds where `rr` does not, we analyze the diversity of happens-before orderings explored by each tool. We instrument both frameworks to record a trace of events per iteration, where each event is identified by a tuple (*source*, *action*, *destination*). Restricting the analysis to the 25 event types, we compute which of the $25 \times 25 - 25 = 600$ ordered event pairs (A, B) with $A \neq B$ each framework explores across all 1,000 iterations: we say a pair is *explored* if there exists at least one iteration in which A occurs before B .



Figure 5.5: Categorical heatmap of happens-before pairs explored by each tool. Orange cells indicate pairs explored only by DIORAMA; dark grey cells indicate pairs explored by both tools; light grey cells indicate pairs explored by neither tool. No pairs are explored exclusively by `rr` (blue). DIORAMA explores a strict superset of the orderings that `rr` reaches.

Table 5.1 and Figure 5.5 summarize the results. Of the 600 ordered pairs, 385 are explored by both tools, 47 are explored exclusively by DIORAMA, and *zero* are explored exclusively by `rr`. The remaining 168 pairs are not explored by either tool—these correspond to orderings that violate the program’s causal structure (e.g., a plan’s deletion cannot precede an event that only occurs during startup) and serve as a sanity check.

Characterizing the additional orderings. The 47 DIORAMA-only pairs are not randomly distributed. They are dominated by orderings in which late-cycle state events of an earlier plan (e.g., `apply/store-fetched`, `apply/endpoints-updated`, `apply/version-updated`, and the corresponding `delete` events) occur *before* early-cycle message events of

a later plan directed at a different Enactor (e.g., `publish-plan`, `fetch-plan`, `register-plan`, `apply-plan`). In other words, DIORAMA discovers interleavings where one plan’s completion and cleanup on one Enactor races with the next plan’s propagation through another Enactor—precisely the cross-Enactor pipeline interleaving that triggers the bug. The OS scheduler used by `rr` consistently serializes these two phases, never exposing the critical window.

These results demonstrate DIORAMA’s ability to systematically explore the joint space of thread interleavings, message orderings, and timing behaviors to discover bugs that arise from rare cross-node interactions—the type of bug that caused a multi-hour outage affecting one of the world’s largest cloud services.

5.4 Use DIORAMA to Analyze Real-World Distributed Systems

The preceding case study demonstrates DIORAMA’s effectiveness on a simplified model of the AWS Route53 incident. We now evaluate whether DIORAMA can scale to *unmodified*, production-grade distributed systems. This section describes how we applied DIORAMA to five widely deployed Java-based distributed systems: Apache ZooKeeper [32], Hazelcast, Apache Ratis, JGroups, and Apache Ignite. We first describe the manual effort required to adapt ZooKeeper and then show how LLM-based agents can automate the process of writing test harnesses for additional systems.

5.4.1 Adapting Real-World Systems to DIORAMA

Running a production distributed system inside a DIORAMA microcosm requires addressing two concerns: (1) the system must be configured so that all communication flows through DIORAMA’s virtualized network and time layers, and (2) a test harness must orchestrate node lifecycles, inject workloads, and assert correctness invariants.

5.4.2 Manual Test Development: Apache ZooKeeper

We chose Apache ZooKeeper [32] as our primary target because it is a widely deployed coordination service with a complex consensus protocol (ZAB), rich client semantics (ephemeral nodes, watches, sequential znodes), and a substantial history of concurrency-related bugs.

Test harness structure. Our ZooKeeper integration test suite contains over 30 test methods organized into six categories:

1. **Leader election and convergence:** tests that verify exactly one leader is elected after startup, cascading failures, and rapid leader churn. These tests target known issues such as ZOOKEEPER-1419, where a 5-node cluster could fail to converge under specific election message orderings.
2. **Split-brain detection:** tests that assert at most one leader exists at any point during network partitions and node failures, targeting the fundamental safety invariant of the ZAB protocol.

3. **Data consistency and replication:** tests that write data through one server and read through all replicas, asserting that committed values are never lost or divergent. These tests cover both normal operation and scenarios with concurrent leader failovers.
4. **Ephemeral node lifecycle:** tests that verify ephemeral znodes are correctly cleaned up when sessions expire or leaders change, targeting bugs where ephemeral nodes “leak” across failovers.
5. **Transaction durability:** tests targeting write-loss scenarios during in-flight proposals, TRUNC sync race conditions when an old leader must truncate its log after failover, and snapshot corruption under write pressure.
6. **Dynamic reconfiguration:** tests that exercise leader demotion, observer promotion, and role changes during concurrent client operations.

Test execution pattern. Each test follows a common structure illustrated by the following annotated pattern:

```

1  @FrayTest(
2      iterations = 1,
3      networkDelegateType = NetworkDelegateType.REACTIVE,
4      systemTimeDelegateType = SystemTimeDelegateType.MOCK,
5      scheduler = POSScheduler.class,
6      abortThreadExecutionAfterMainExit = true
7  )
8  void testPotentialWriteLoss() throws Exception {
9      // Phase 1: Launch cluster and wait for stable state
10     setupZookeeperServers(numServers);
11     for (int i = 1; i <= numServers; i++)
12         serverYarns.add(launchZookeeperServer(configPath));
13     Thread.sleep(300000L); // Logical time: no real delay
14
15     // Phase 2: Inject failure
16     int leaderId = findLeader(numServers);
17     stopYarn(serverYarns.get(leaderId - 1));
18
19     // Phase 3: Observe and assert invariant
20     Map<Integer, ProbeDataState> states =
21         queryPathDataByServer(numServers, "/test-path");
22     for (ProbeDataState state : states.values())
23         assertEquals(expected, state.value(),
24             "Replica divergence detected");
25 }

```

The `@FrayTest` annotation configures DIORAMA’s scheduler (here, partial-order sampling [78]), enables the reactive network delegate for message-order exploration, and activates mock system time. The `abortThreadExecutionAfterMainExit` flag ensures that lingering server threads are terminated between iterations.

Known bugs targeted. Our test suite targets several known ZooKeeper bugs that arise from specific thread and message interleavings:

- ZOOKEEPER-1419: leader election non-convergence in 5-node clusters under specific message orderings during cascading failures.
- ZOOKEEPER-1492: a leader continuing to serve after all followers have disconnected, violating the quorum safety invariant.
- ZOOKEEPER-4846: ACL map corruption caused by a race between fuzzy snapshot writes and concurrent client operations.
- ZOOKEEPER-4306: instability triggered by large close-session transactions that interact with snapshot boundaries.

These bugs require precise control over the interleaving of election messages, proposal broadcasts, and session management events. In our evaluation, DIORAMA successfully identified and reproduced all of these bugs without any manual intervention.

5.4.3 LLM-Assisted Test Generation

Manually writing integration tests for each target system requires deep familiarity with its architecture, protocol, and configuration. To evaluate whether this process can be automated, we designed an LLM-based agent workflow that takes the ZooKeeper test suite as a reference and generates analogous test harnesses for new systems.

Agent design. The agent operates in six phases, each producing structured output that feeds into the next:

1. **System selection:** the agent identifies a candidate distributed system based on criteria including popularity, Java implementation, consensus/replication complexity, and feasibility under DIORAMA’s execution model.
2. **Pattern extraction:** the agent analyzes the ZooKeeper test suite to extract reusable patterns—the `ServerManager` interface, thread-pool minimization, predicate-based waiting, multi-probe assertion, and result communication via system properties.
3. **Bug hypothesis generation:** the agent studies the target system’s source code and documentation to hypothesize concurrency-sensitive scenarios, focusing on protocol-level behaviors such as state divergence, inconsistent leader election outcomes, lost updates, and ordering violations.
4. **Test implementation:** the agent produces `ServerManager` implementations for each node type and a test harness that orchestrates the scenario. All tests follow the three-phase structure: setup, chaos injection, and invariant assertion.
5. **Validation:** the agent executes the test under DIORAMA and classifies failures, filtering out client-side artifacts, timeout-only failures, and non-reproducible issues.
6. **Refinement:** based on validation results, the agent iterates on the test design, adjusting thread-pool sizes, timing parameters, or the target invariant.

Table 5.2: Target systems and the concurrency properties tested by the LLM-generated harnesses.

System	Concurrency Model	Properties Tested
Hazelcast	Distributed map with partition routing	Linearizability, atomic updates
Apache Ratis	Raft consensus with state machine	Linearizable reads, replication
JGroups	Group communication (multicast)	FIFO ordering, total ordering
Apache Ignite	Transactional distributed cache	Serializable isolation

Focus on server-side invariants. The agent is explicitly constrained to target server-side safety, liveness, and consistency violations rather than client-observed anomalies. A valid bug must manifest as a violated server-side invariant (e.g., replica divergence, split-brain, lost committed write) that can be detected by probing the servers directly after the test scenario completes.

5.4.4 LLM-Generated Test Suites

Using the agent workflow described above, we generated test harnesses for four additional distributed systems. Table 5.2 summarizes the target systems and the properties tested.

Hazelcast. Hazelcast is a distributed in-memory data grid that partitions data across cluster members. The generated test suite includes two configurations: a single-node test that exercises internal concurrency within Hazelcast’s operation threads, and a two-node cluster test that forces operations through the distributed protocol (partition routing, backup replication, and lock coordination). Both configurations test a shared counter increment under three synchronization modes: distributed locks, compare-and-swap (CAS), and unsynchronized access. The unsynchronized mode serves as a positive control—DIORAMA should detect lost updates, confirming that the scheduling exploration is effective. The test reduces Hazelcast’s thread pools to a single thread each (I/O, operation, generic operation, event, and response threads) and disables diagnostics and metrics to minimize the schedule space.

Apache Ratis. Apache Ratis is a Java implementation of the Raft consensus protocol [57]. The generated test wraps each Ratis server as a `ServerManager` with a counter-based state machine that supports `increment` and `get` operations. The test launches a three-node Raft group using Netty transport with `epoll` disabled for DIORAMA compatibility. A client issues interleaved write and linearizable-read operations, recording the operation history. The test oracle verifies that linearizable reads always observe the most recently committed write, detecting any violations of Raft’s linearizability guarantee. A separate probe client queries server roles to validate that at most one leader exists at any time.

JGroups. JGroups is a group communication library used by many Java middleware systems (WildFly, Infinispan, Keycloak). The generated tests exercise three scenarios:

- **FIFO ordering:** multiple nodes send numbered messages via reliable multicast (NAKACK2 protocol), and the test verifies that messages from each sender are received in send order at all nodes.
- **Total ordering:** the same scenario with the SEQUENCER protocol added, verifying that all nodes receive all messages in the same global order.
- **Eager send:** nodes send messages immediately after connecting without waiting for a stable view, stressing the interaction between view installation (GMS protocol) and message delivery (NAKACK2), which is a known source of message-loss bugs.

Apache Ignite. Apache Ignite provides a distributed SQL-capable cache with ACID transaction support. The generated test exercises Ignite’s transactional cache under concurrent access with three transaction modes: pessimistic (internal read locks prevent conflicts), optimistic (conflicts detected at commit time with automatic retry), and no-transaction (raw get/put, expected to lose updates). The test validates that the final counter value equals the expected sum of all increments, detecting atomicity violations in the transaction protocol.

5.4.5 Limitations and Future Work

Network mocking and the applicability dilemma. In Chapter 3, we discussed that replacing concurrency primitives with mocked implementations limits applicability due to complex interactions between the JVM runtime and application code. DIORAMA’s network virtualization encounters a similar challenge. First, our instrumentation intercepts operations on `Socket` and `ServerSocket` but does not correctly handle application classes that *subclass* these types. Second, DIORAMA does not support Netty, a widely adopted networking library that bypasses the Java networking abstraction entirely by invoking native system calls (via JNI) to interact with the OS network stack through `epoll`, `kqueue`, or `io_uring`. Because our interception operates at the Java API boundary, Netty’s native call paths are invisible to the instrumentation. This required us to disable Netty’s native transport and fall back to JDK NIO in systems such as Apache Ratis (§5.4.4).

Graceful shutdown. Most distributed systems are designed to run indefinitely and do not expose a clean shutdown interface. Because DIORAMA executes multiple test iterations within the same JVM process, it must fully terminate and reclaim all resources from one iteration before starting the next. In practice, many systems spawn daemon threads, register shutdown hooks, or hold resources that outlive the logical node’s lifecycle. To address this, DIORAMA provides an `abortThreadExecutionAfterMainExit` option that forcibly terminates lingering threads by injecting exceptions once the main thread exits. While effective at preventing cross-iteration interference, this mechanism can itself trigger spurious errors during shutdown or cause the test process to hang when threads catch and suppress the injected exceptions.

One promising direction for improving shutdown reliability is to fork a fresh process for each test iteration rather than reusing a single JVM. Under this model, each iteration runs in an isolated child process that the parent can terminate unconditionally via an OS-level kill signal, eliminating the need for cooperative in-process shutdown entirely. Because process termination is handled by

the operating system, lingering daemon threads, registered shutdown hooks, and leaked resources are all reclaimed regardless of application behavior. The primary cost is the JVM startup overhead incurred per iteration; however, this can be amortized using checkpoint-restore mechanisms such as CRaC [3] or GraalVM native images, which allow a pre-initialized JVM snapshot to be restored in milliseconds.

Other sources of nondeterminism. FRAY guarantees deterministic record and replay under the assumption that concurrency is the sole source of nondeterminism in the target program. This assumption holds for most unit-level concurrency tests, but breaks down when testing full distributed systems. Production systems routinely depend on additional nondeterministic inputs: `Object.hashCode()` (which defaults to an identity hash derived from memory layout), random number generators used for jitter, UUID generation, and load balancing, as well as file-system ordering, environment variables, and locale-sensitive formatting. Because DIORAMA does not currently intercept or control these sources, two iterations with identical thread and message schedules may still diverge in behavior, preventing faithful replay of a failure-inducing execution.

Chapter 6

Conclusion and Discussion

6.1 Summary

This thesis argues that effective concurrency testing does not require faithfully reproducing the exact execution environment of the target program—what matters is preserving and exploring the space of observable program behaviors. Guided by this behavior-centric principle, we built three systems: FRAY, which serializes thread execution via shadow locking to enable deterministic concurrency control on real-world JVM programs; FEST, which uses behavioral feedback to adaptively guide schedule exploration; and DIORAMA, which reduces distributed system testing to single-process concurrent testing by virtualizing the network and time. Together, they demonstrate that decoupling testing from faithful execution yields control, efficiency, and effectiveness simultaneously—without sacrificing applicability to real-world software.

6.2 Discussion: Where Are We Heading?

When I began this PhD, large language models were not available as practical tools. Today, they write compilers from scratch. The trajectory is unmistakable: we are entering an era of unprecedented code-generation power, where the bottleneck in building software is shifting from writing code to understanding whether that code is correct.

This asymmetry should concern us. The capacity to produce software is growing far faster than the capacity to validate it. A developer can generate thousands of lines of concurrent, distributed code in minutes. But our ability to reason about the correctness of that code, to find the subtle interleaving that triggers a deadlock or the rare message ordering that corrupts state, has not kept pace. The gap between code production and code validation is widening.

I believe this gap represents one of the most important challenges for the next generation of software engineering research. Three trends make this especially urgent.

Software is becoming more concurrent and distributed. The era of single-threaded, single-machine software is long over. Modern applications are concurrent by default: they use asynchronous frameworks, parallel data pipelines, and microservice architectures. Cloud-native systems distribute computation across dozens or hundreds of nodes, each running multithreaded

code that communicates over unreliable networks. Every additional thread, message channel, and timer multiplies the space of possible behaviors. The combinatorial explosion that makes concurrency testing hard today will only intensify.

Developers will expect more properties from their software. As software takes on greater responsibility—managing critical infrastructure, handling financial transactions, making autonomous decisions—the bar for correctness will rise. It will no longer be sufficient to test that a system handles the happy path. Developers and users will demand guarantees about fairness, consistency, liveness, and graceful degradation under failure. Each of these properties interacts with concurrency in complex ways, and testing for them requires exploring behaviors that are exponentially rare under normal execution.

AI-generated code amplifies the need for testing. When code is written by an LLM, the developer may not fully understand every line. The implicit knowledge that a human programmer carries—awareness of subtle invariants, corner cases learned from debugging sessions, intuition about which interleavings are dangerous—may not transfer to generated code. This makes rigorous testing not merely important but essential: it becomes the primary mechanism through which developers build confidence in code they did not write themselves.

These trends point to a future where testing infrastructure is at least as important as code-generation infrastructure. The systems presented in this thesis take steps in this direction: FRAY makes controlled concurrency testing practical for real-world JVM programs; FEST makes schedule exploration adaptive and efficient; DIORAMA extends these capabilities to distributed systems. But much work remains.

Bibliography

- [1] Java Concurrency Stress (jstress). <https://github.com/openjdk/jcstress>, . 2.3
- [2] Java micro-library for writing & testing concurrent code. <https://tempusfugitlibrary.org/>, . 2.3
- [3] CRaC: Coordinated restore at checkpoint. <https://crac.github.io/use/crac-runtime/index.html>, 2024. Accessed: 2026-05-15. 5.4.5
- [4] Fire-flyer file system. <https://github.com/deepseek-ai/3FS/tree/main/specs>, 2024. 4, 4.1.1, 4.1, 4.4.2
- [5] Guava: Google core libraries for java. <https://github.com/google/guava>, 2024. 3, 3.2
- [6] Kafka streams. <https://kafka.apache.org/documentation/streams/>, 2024. 3, 3.2
- [7] Apache lucene. <https://lucene.apache.org/>, 2024. 3, 3.2
- [8] Amazon Web Services. Summary of the AWS Service Event in the US East (N. Virginia) Region. <https://aws.amazon.com/message/101925/>, 2025. 5.1, 5.1.1, 5.3.1
- [9] Apache Flink. FlinkCEP - Complex event processing for Flink. <https://nightlies.apache.org/flink/flink-docs-master/docs/libs/cep/>, 2024. 4.3.5
- [10] Baeldung. Testing multi-threaded code in Java. <https://www.baeldung.com/java-testing-multithreaded>, 2024. May 2024 version archived at <https://web.archive.org/web/20240515050437/https://www.baeldung.com/java-testing-multithreaded>. 1
- [11] Arpit Bhayani. Heartbeats in distributed systems. <https://arpitbhayani.me/blogs/heartbeats-in-distributed-systems/>, 2024. Accessed: 2026-05-01. 5.1.2, 5.1.2
- [12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 1032–1043, 2016. doi: 10.1145/2976749.2978428. 4, 4.2.1
- [13] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. *Commun. ACM*, 66(11):89–97, oct 2023. ISSN 0001-0782. doi: 10.1145/3611019. URL <https://doi.org/10.1145/3611019>. 4.2.2

- [14] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21*, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483540. URL <https://doi.org/10.1145/3477132.3483540>. 4
- [15] Marc Brooker and Ankush Desai. Systems correctness practices at amazon web services. *Commun. ACM*, 68(6):38–42, June 2025. ISSN 0001-0782. doi: 10.1145/3729175. URL <https://doi.org/10.1145/3729175>. 4
- [16] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGPLAN Not.*, 45(3):167–178, March 2010. ISSN 0362-1340. doi: 10.1145/1735971.1736040. URL <https://doi.org/10.1145/1735971.1736040>. 2.4, 3, 3.1, 3.1.3, 4, 7, 4.4.2
- [17] Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. Industrial-strength controlled concurrency testing for programs with Coyote. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 433–452. Springer, 2023. 2.4, 4, 7, 4.4.2
- [18] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe asynchronous event-driven programming. *SIGPLAN Not.*, 48(6):321–332, jun 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462184. URL <https://doi.org/10.1145/2499370.2462184>. 4.1.1, 4.4.1, 4.1, 4.4.2
- [19] Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, page 73–83, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786861. URL <https://doi.org/10.1145/2786805.2786861>. 2.4, 4.1.1, 7
- [20] Ankush Desai, Sanjit A. Seshia, Shaz Qadeer, David Broman, and John C. Eidson. Approximate synchrony: An abstraction for distributed almost-synchronous systems. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 429–448. Springer, 2015. doi: 10.1007/978-3-319-21668-3_25. URL https://doi.org/10.1007/978-3-319-21668-3_25. 4.1.1
- [21] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.*, 2 (OOPSLA), oct 2018. doi: 10.1145/3276529. URL <https://doi.org/10.1145/3276529>. 4.1.1, 4.4.2
- [22] Ankush Desai, Shromona Ghosh, Sanjit A. Seshia, Natarajan Shankar, and Ashish Tiwari. Soter: A runtime assurance framework for programming safe robotics systems. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*,

- pages 138–150, 2019. doi: 10.1109/DSN.2019.00027. 4.1.1
- [23] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pages 411–422, 2011. 3.1.3
 - [24] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP ’03, page 237–252, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137575. doi: 10.1145/945445.945468. URL <https://doi.org/10.1145/945445.945468>. 1.1
 - [25] EsperTech. Complex Event Processing, Streaming SQL and Event Series Analysis for Java. <https://github.com/espertechinc/esper>, 2024. 4.3.5
 - [26] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’05, page 110–121, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 158113830X. doi: 10.1145/1040305.1040315. URL <https://doi.org/10.1145/1040305.1040315>. 1.1, 2.4, 7, 21
 - [27] GitHub. The top programming languages. <https://octoverse.github.com/2022/top-programming-languages>, 2022. 1
 - [28] Google. This is weaver, a framework for writing multi-threaded unit tests in java. <https://github.com/google/thread-weaver>, 2024. 2.3
 - [29] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java® Language Specification—Java SE 22 Edition*, chapter 17.4, pages 765–780. Oracle America, Inc., 2024. 1.1, 3.1.1, 3.1.2
 - [30] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The /spl phi/ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pages 66–78, 2004. doi: 10.1109/RELDIS.2004.1353004. 5.1.2
 - [31] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. 1.2.1
 - [32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIX ATC’10. USENIX Association, 2010. 5.4, 5.4.2
 - [33] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Rosu, and Darko Marinov. Improved multithreaded unit testing. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE ’11, page 223–233, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. doi: 10.1145/2025113.2025145. URL <https://doi.org/10.1145/2025113.2025145>. 2.3
 - [34] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security*

and Privacy (SP), pages 754–768. IEEE, 2019. 2.4

- [35] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022. 2.4
- [36] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*, pages 675–681. Springer, 2009. 2.3, 3
- [37] Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. Lincheck: A practical framework for testing concurrent data structures on JVM. In *International Conference on Computer Aided Verification*, 2023. 1.2.1, 2.3, 3
- [38] Elastic Search Labs. Concurrency bugs in lucene: How to fix optimistic concurrency failures. <https://www.elastic.co/search-labs/blog/concurrency-bugs-lucene-debugging>, 2025. 3
- [39] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL <https://doi.org/10.1145/359545.359563>. 4, 10, 4.4.2
- [40] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, may 1994. ISSN 0164-0925. doi: 10.1145/177492.177726. URL <https://doi.org/10.1145/177492.177726>. 1.2.2
- [41] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa>. 2.1, 2.5, 5
- [42] Ao Li, Byeongjee Kang, Vasudev Vikram, Isabella Laybourn, Samvid Dharanikota, Shrey Tiwari, and Rohan Padhye. Fray: An efficient general-purpose concurrency testing platform for the jvm. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025. doi: 10.1145/3764119. URL <https://doi.org/10.1145/3764119>. 1.2.1, 3
- [43] Ao Li, Ankush Desai, and Rohan Padhye. Feedback-guided adaptive testing of distributed systems designs. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pages 701–719, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. URL <https://www.usenix.org/conference/nsdi26/presentation/li>. 1.2.2, 4
- [44] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 162–180, 2019. 1.2.1, 2.4

- [45] Lun Liu, Todd Millstein, and Madanlal Musuvathi. A volatile-by-default JVM for server applications. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133873. URL <https://doi.org/10.1145/3133873>. 1.1
- [46] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008. 1, 1.1, 3.2, 5.1.1
- [47] Shan Lu, Soyeon Park, and Yuanyuan Zhou. Detecting concurrency bugs from the perspectives of synchronization intentions. *IEEE Transactions on Parallel and Distributed Systems*, 23(6):1060–1072, 2011. 1.1
- [48] LUCENE-10112. Lucene-10112. <https://github.com/apache/lucene/issues/10112>. Accessed: 2024-11-20. 1
- [49] LUCENE-13127. Lucene-13127. <https://github.com/apache/lucene/issues/13127>. Accessed: 2024-11-20. 1
- [50] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys ’19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. doi: 10.1145/3302424.3303986. URL <https://doi.org/10.1145/3302424.3303986>. 2.1, 2.5, 5
- [51] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’23, page 1615–1629, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi: 10.1145/3576915.3623097. URL <https://doi.org/10.1145/3576915.3623097>. 2.1, 2.5, 4, 4.1.4, 10, 4.4.2, 5
- [52] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. Learning-based controlled concurrency testing. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428298. URL <https://doi.org/10.1145/3428298>. 4, 4.4.2
- [53] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices*, 42(6):446–455, 2007. 3.1.3
- [54] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtii. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, 2008. 1, 2.4, 7
- [55] Robert O’callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, 2003. 1.1
- [56] Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Par-tush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical*

- Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. URL <https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan>. 3, 4, 15, 4.2.2, 5.3, 5.3.2
- [57] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC’14, pages 305–319. USENIX Association, 2014. 5.4.4
- [58] Oracle. Package `java.util.concurrent.atomic`. <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/atomic/package-summary.html>. 3.1.1
- [59] P Organization. P - Case Studies. <https://p-org.github.io/P/casestudies/>, 2024. 4.1.1
- [60] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330576. URL <https://doi.org/10.1145/3293882.3330576>. 4, 4.2.2
- [61] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360600. URL <https://doi.org/10.1145/3360600>. 4.2.2
- [62] Lauren Pick, Ankush Desai, and Aarti Gupta. Psym: Efficient symbolic exploration of distributed systems. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023. doi: 10.1145/3591247. URL <https://doi.org/10.1145/3591247>. 4.1, 4.4.2
- [63] David Schwartz, Ankith Kowshik, and Luís Pina. Jmvx: Fast multi-threaded multi-version execution and record-replay for managed languages. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi: 10.1145/3689769. URL <https://doi.org/10.1145/3689769>. 2.3, 3
- [64] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008. 1.1
- [65] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009. 1.1
- [66] Stack Overflow. Developer survey—most popular technologies: Programming, scripting, and markup languages. <https://survey.stackoverflow.co/2023/#technology-most-popular-technologies>, 2023. 1
- [67] Paul Thomson. *Practical systematic concurrency testing for concurrent and distributed software*. PhD thesis, Imperial College London, 2016. 2.3

- [68] Paul Thomson, Alastair F Donaldson, and Adam Betts. Concurrency testing using controlled schedulers: An empirical study. *ACM Transactions on Parallel Computing (TOPC)*, 2(4): 1–37, 2016. 3, 3.1.3
- [69] TIOBE Software. TIOBE index. <https://www.tiobe.com/tiobe-index/>, 2024. July 2024 Index archived at <https://web.archive.org/web/20240709010804/https://www.tiobe.com/tiobe-index/>. 1
- [70] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In Nigel Davies, Seitz Jochen, and Kerry Raymond, editors, *Middleware'98*, pages 55–70, London, 1998. Springer London. ISBN 978-1-4471-1283-9. 5.1.2
- [71] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated software engineering*, 10:203–232, 2003. 2.3, 2.4, 3
- [72] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. *Sun Microsystems Laboratories*, 1994. 5.1.2
- [73] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. Controlled concurrency testing via periodical scheduling. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*, 2022. ISBN 9781450392211. doi: 10.1145/3510003.3510178. 2.4
- [74] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI'09*, page 213–228, USA, 2009. USENIX Association. 2.1, 2.5, 5
- [75] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 485–502, 2012. 3.1.3
- [76] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. DDRace: Finding concurrency UAF vulnerabilities in linux drivers with directed fuzzing. In *USENIX Security Symposium*, pages 2849–2866, 2023. 2.4
- [77] Xinhao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20*, page 1141–1156, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378484. URL <https://doi.org/10.1145/3373376.3378484>. 2.4, 4, 7, 18, 4.4.2
- [78] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 317–335. Springer, 2018. doi: 10.1007/978-3-319-96142-2_20. URL https://doi.org/10.1007/978-3-319-96142-2_20. 2.4, 3, 3.1, 3.1.3,

4, 21, 18, 4.4.2, 5.4.2