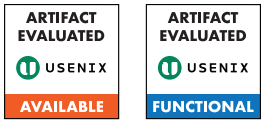

Feedback-guided Adaptive Testing of Distributed Systems Designs

Ao Li[1], Ankush Desai[2], Rohan Padhye[1]
[1]*Carnegie Mellon University,* [2]*Amazon Web Services*

Abstract

Validating distributed systems for correctness poses significant challenges. Practitioners often rely on formal models of core system designs, which are then tested by exploring possible component interactions. Unfortunately, standard testing approaches based on random sampling of the state space are inefficient and prone to missing subtle bugs, as they lack guidance from the system's behavior.

To address this, we present Fest, a new testing system for formal models of distributed systems. Fest incorporates feedback-guided adaptive schedule generation, drawing inspiration from grey-box fuzzing, to steer exploration towards maximizing behavioral coverage and uncovering bugs more effectively. Our implementation in the P programming framework demonstrates significant improvements across 94 distributed system model configurations: up to 41× (1.5× average) improvement in behavioral coverage, 278× (15× average) improvement in scenario coverage, and 33% more bugs detected compared to existing methods. These results highlight Fest's effectiveness in ensuring the robustness of distributed systems through improved testing efficiency.

1 Introduction

Distributed systems underpin critical cloud infrastructure and services, demanding correctness assurance in both design and implementation. To avoid costly refactoring and redesign, leading practitioners such as Amazon Web Services (AWS) and DeepSeek adopt a *model-first* approach: they implement core algorithms in modeling languages like P [18] and TLA+ [30], which enable verification and testing of designs before any code is written.

While exhaustive state-space exploration provides the strongest correctness guarantees, it becomes computationally intractable for complex, real-world systems. A widely adopted and practical alternative, particularly in industry, is *model-based design testing*. In this approach, developers model core system algorithms using high-level specification languages and then employ lightweight formal methods [10, 11], which amount to *randomized testing* of distributed systems, to uncover potential issues. This lightweight, bug-finding approach trades exhaustive guarantees for practical scalability, revealing subtle design flaws before they escalate into costly implementation bugs.

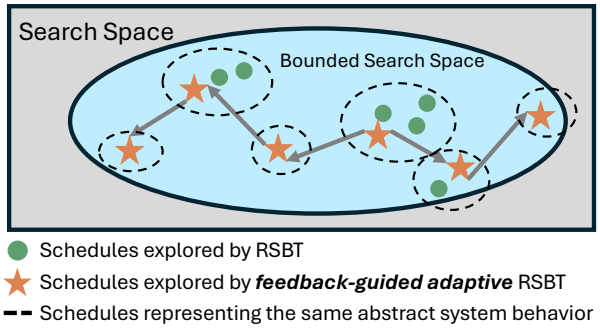

Figure 1: Overview of the search performed by RSBT and feedback-guided adaptive RSBT. Each edge connects two schedules, indicating that the target schedule is generated adaptively based on the source schedule.

Design testing frameworks typically model distributed systems as communicating state machines. The testing tool controls the message ordering, exploring system behavior under different schedules (sequences of events). The effectiveness of testing hinges on the scheduling algorithm used to generate these schedules from the vast search space of possible event interleavings. A key objective for scheduling algorithms is maximizing behavior coverage—exploring as many unique system states and transitions as possible within a given time budget—as exhaustive exploration is infeasible.

Current state-of-the-art techniques are often variants of *randomized search-based bounded concurrency testing (RSBT)* [12, 56, 57], such as Probabilistic Concurrency Testing (PCT) [12] and Partial Order Sampling (POS) [57]. These algorithms prune the search space using scheduling constraints (e.g., limiting context switches [12] or focusing on partial-order dependencies [57]), focusing exploration on schedules deemed more likely to expose unique behaviors or

bugs (visualized by the blue ellipse in Figure 1). This constrained approach has proven effective in practice for finding bugs compared to unconstrained random exploration.

Existing RSBT algorithms suffer from a fundamental limitation: they rely primarily on random sampling within the constrained space. Randomness, akin to black-box fuzzing [8], 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) [39]—which learn scheduling actions based on system state and rewards—seem a natural solution, they have proven surprisingly ineffective. Our experiments (Section 5.3), consistent with prior studies [17], show QL performing even worse than RSBT methods on complex systems. This underscores the value of RSBT's constraints but highlights that effective exploration within these constraints requires a different approach than random sampling.

This paper proposes integrating feedback-guided adaptive schedule generation into RSBT algorithms, replacing ineffective random sampling. Inspired by the success of grey-box fuzzing [8] in software testing, our goal is to leverage information from executed schedules to intelligently guide the generation of subsequent schedules towards unexplored behaviors, while respecting the beneficial constraints of RSBT.

However, applying this idea to distributed system testing presents two key challenges: (1) Directly mutating a schedule, such as altering the execution order between machines, can produce infeasible schedules, such as causing a machine to become unschedulable (e.g., being blocked from receiving a necessary message). Additionally, such mutations may violate the scheduling constraints of RSBT algorithms, for instance, by exceeding the permitted number of context switches. (2) Given the vast search space of possible schedules, the generation algorithm must effectively identify and prioritize schedules that are more likely to uncover unique behaviors.

This paper presents FEST, the first distributed design testing technique aimed at maximizing behavioral coverage. FEST addresses the challenges above with two innovative designs. First, FEST integrates record-and-replay [16, 41] techniques with parametric input generation [46]. It records the execution of RSBT algorithms and applies small, controlled mutations to the recorded execution. These mutated recordings are then replayed to generate new schedules that are similar yet distinct from the original, facilitating adaptive schedule generation. To tackle the second challenge—identifying schedules likely to uncover unique behaviors—FEST introduces a *diversity feedback* mechanism that evaluates the uniqueness of a schedule's behavior. This evaluation is based on abstract Lamport timelines [29, 38], 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 the arrows and stars in Figure 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 [1], 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× (1.5× on average) more behavioral coverage, 278× (15× 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× improvement in timeline coverage.

In this paper, we make the following key contributions:

1. We introduce a novel feedback-guided adaptive schedule generation algorithm for distributed system models designed to maximize behavioral coverage of existing state-of-the-art randomized search-based bounded concurrency testing algorithms. Our implementation is now part of the official P repository and is used by the P community. The artifact of FEST is available at: https://github.com/aolial/Fest-artifact
2. We introduce a scenario specification language enabling developers to optionally define critical and corner-case scenarios, which the feedback algorithm then uses to guide the fuzzer in generating schedules that satisfy these scenarios.
3. Our extensive evaluation includes 19 distributed systems modeled in P, covering 94 distinct configurations, and demonstrated significant improvements in behavioral coverage, scenario coverage, and bug detection capabilities.

2 Background

2.1 P Framework

P [6, 18, 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 [48], to verify that the system model satisfies its desired specifications. P is used across industry (e.g., AWS [5], Microsoft [18], and

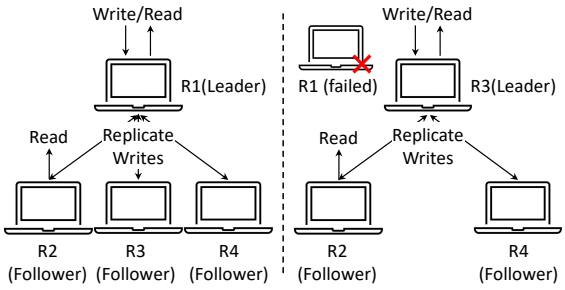

Figure 2: A distributed storage system with four replicas.

DeepSeek [1]) and academia [20, 21, 22] for reasoning about distributed protocols. At AWS, we use P for reasoning about the correctness of core distributed protocols driving our services across compute, databases, and storage systems.

Replicated storage system modeled in P. We consider a simple replicated key-value store consisting of one leader and three follower replicas (Figure 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 2 right side).

Figure 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.

2.2 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 [12]. Partial order sampling (POS) applies partial order

```
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 3: Replica state machine in P.

Algorithm 1: The testing loop using an RSBT algorithm.

```
1  Function RSBTTesting(Program P, RSBT algorithm A):
2      repeat
           // Sample a new schedule S.
3          S ← A(P)
           // Get execution trace E.
4          E ← Execute(P, S)
5          if E violates specification assertions then
6              raise error(E)
7      until out of time budget
```

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

reduction [24] 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 [12, 17, 19, 40] and have helped uncover real-world bugs in concurrent [12, 40], distributed systems [17, 56].

Algorithm 1 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.

2.3 Deterministic Record and Replay

Record-and-replay techniques have been successful in debugging complex software [16, 41]. 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 2 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`)`.

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

2.4 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 [38]. Recently, researchers have proposed to measure behavioral coverage in distributed systems via *abstract Lamport timelines* (*timeline coverage*) [38]. Timelines capture the happens-before relationships between events or messages exchanged. In this paper, 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.

3 Problem and Solution Overview

3.1 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 [8].

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.

3.2 Feedback-Guided Adaptive Scheduling

In this work, 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 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 [16, 41] and parametric generators [46]. 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 [9, 47]. 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.

3.3 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 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 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 4.5. 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.

3.4 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.

4 FEST Design

4.1 Main Algorithm

Algorithm 3 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 4.2.

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 4.4). 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 4.6. Otherwise, it sets *compliance* to 1 (Line 23), indicating that all schedules have the same compliance score.

Algorithm 3: 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 ← Record(P, A)
3      ℝ ← {R}
4      𝕊, 𝔼 ← ∅
5      repeat
6          R ← ℝ.Dequeue()
7          repeat
8              // Adaptive Schedule Generation (§4.2)
9              R' ← Mutate(R)
10             S' ← Replay(P, A, R')
11             // System Execution (§4.3)
12             E ← Execute(P, S')
13             if E violates specification assertions then
14                 raise error(E)
15             // Diversity Feedback (§4.4)
16             diversity ← Diversity(𝔼, E)
17             if diversity = 0 then
18                 continue
19             // Compliance Feedback (§4.6)
20             if scen is provided then
21                 compliance ← Compliance(scen, E)
22             else
23                 compliance ← 1
24             𝔼 ← 𝔼 ∪ {E}
25             𝕊 ← 𝕊 ∪ {S'}
26             ℝ.Insert(R', diversity * compliance)
27         until out of exploration budget
28     until out of time budget
29     return 𝕊
```

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).

4.2 Adaptive Schedule Generation

A *schedule* S is defined as a sequence of machine interleavings $m_1, m_2, \ldots$ 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 immedi-

Algorithm 4: The PCT algorithm. The function calls to pseudo-randomness is highlighted in blue.

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

ately after $\bar{S}$. We denote the set of such *enabled* machines when program P is executed with schedule prefix $\bar{S}$ as `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 archive this, FEST mutates an average of five integers per recording.

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

Partial Order Sampling Algorithm 5 presents the schedule generation algorithm that employs partial order sampling [57]. 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 [24] 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 [57] algorithm with conflict analysis [56]. Section 5.3 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 System Execution

Back to Algorithm 3, 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 evaluate the feedback of generated schedule S.

Algorithm 5: The POS algorithm. The function calls to pseudo-randomness is highlighted in blue.

```
1  Function POS(Program P):
2      Q ← PriorityQueue(initial = {entryMachine})
3      S ← ∅
4      repeat
5          for idx ∈ {0,...,|Q|−1} do
6              if Q[[idx]] ∈ Enabled(P, S) then
7                  m ← Q[[idx]]
8                  break
           // Reset priorities
9          for m′ ∈ Enabled(P, S) do
10             if IsRacing(m′, m) then
11                 Q ← Q\{m′}
12         S ← S m
13         for m′ ∈ Enabled(P, S) do
14             if m′ ∉ Q then
15                 prio ← RandInt(|Q|)
16                 Q.Insert(prio, m′)
17     until enP = ∅
18     return S
```

Algorithm 6: Diversity feedback algorithm.

```
1  Function ToTimeline(Execution events E):
2      T ← ∅
3      for e, e′ ∈ E do
4          if e′ ≺ e ∧ e′.receiver = e.receiver then
5              T ← T ∪ {⟨e.receiver, e′.name, e.name⟩}
6      return T
7  Function Diversity(Execution history 𝔼, List of Events E):
8      𝕋 ← ⋃_{E′∈𝔼} ToTimeline(E′)
9      T ← ToTimeline(E)
10     return min_{T′∈𝕋}(1 − J(T, T′))
```

4.4 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 3, 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 [38], 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 [29, 38]. The `ToTimeline` function shown in Algorithm 6 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 6 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 $\mathbb{T}$.

$$
\begin{aligned}
\mathcal{S} &:= \langle name\rangle(\mathsf{Event}(,\mathsf{Event})^*): \\
&\quad \mathcal{C}(\wedge\mathcal{C})^* \\
\mathcal{C} &:= \mathsf{Event}(\prec \mathsf{Event})^+ \\
&\quad \mid \mathcal{V} = \mathcal{V} \\
&\quad \mid \mathcal{V} \neq \mathcal{V} \\
\mathcal{V} &:= \mathsf{Event.name} \\
&\quad \mid \mathsf{Event.sender} \\
&\quad \mid \mathsf{Event.receiver} \\
&\quad \mid \mathsf{Event.}\langle payload\_field\rangle \\
&\quad \mid \langle const\rangle
\end{aligned}
$$

(a) Scenario Language.

$$
\begin{aligned}
&\mathrm{S1}(e_0, e_1): \\
&\quad e_0 \prec e_1 \wedge e_1.\mathtt{key} = e_0.\mathtt{key} \wedge \\
&\quad e_0.\mathtt{name} = \text{"writeReq"} \wedge \\
&\quad e_1.\mathtt{name} = \text{"readReq"} \wedge \\
&\mathrm{S2}(e_0, e_1, e_2, e_3): \\
&\quad e_0 \prec e_1 \prec e_2 \prec e_3 \wedge \\
&\quad e_1.\mathtt{name} = \text{"writeReq"} \wedge \\
&\quad e_3.\mathtt{name} = \text{"readReq"} \wedge \\
&\quad e_3.\mathtt{key} = e_1.\mathtt{key} \wedge \\
&\quad e_0.\mathtt{name} = \text{"newLeaderElected"} \wedge \\
&\quad e_2.\mathtt{name} = \text{"newLeaderElected"} \wedge \\
&\quad e_0.\mathtt{leader} \neq e_2.\mathtt{leader}
\end{aligned}
$$

(b) Examples.

Figure 4: Scenario specification language and example scenarios for the key-value store in § 3.3

4.5 Scenario Specification Language

Drawing inspiration from complex event processing tools like FlinkCEP [3] and Esper [2], we design a scenario specification language to guide FEST toward exploring targeted behaviors.

Figure 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* (Event $\prec$ 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 4b shows the scenarios S1 and S2 described in § 3.3 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.`leader` $\neq e_2$.`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 $e_1, \ldots, e_n$ if and only if there exists a (not necessarily contiguous) subsequence of events $e_1, \ldots, e_n$ in E such that $Scen(e_1, \ldots, e_n)$ is true.

4.6 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. 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 $\frac{3}{4}$.

5 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?

5.1 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 added support for specifying scenarios, (2) extending the existing P checker to collect feedback dynamically, (3) and a feedback-guided adaptive schedule generator for 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.

5.2 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 [12, 17], POS [57], POS with conflict analysis (POS+) [56] and traditional RL based schedule generation (QL) [17, 39] 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 $depth = 3, 15, 50$, 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 in prior works [17, 56]. 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 3, Line 26).

Comparison metric. We use abstract Lamport timeline [29, 38] 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 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, 48], 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 [1], 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 2 and applied these scenarios to 4 models. We present the aggregated results in the following sections, with detailed tables for each benchmark and technique available in Appendix A.

Table 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
Elastic Storage	4539
Leader Election	2748
Stream Log	1495
Memory Database	3219
Network Dynamic Scaling	4295
Network Routing	5392
Database Storage Engine	3955
Access Management	3298
Concurrency Control	3056
FreeRTOS	1485

AWS System	LOC
CDN	1291
Bucket Storage	994
Financial Service	4095
Message Broker	1456

Open-source System	LOC
German [48]	282
OSR [48]	381
Paxos [48]	598
2PhaseCommit [18]	822
DeepSeek-3FS [1]	4648

Table 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.

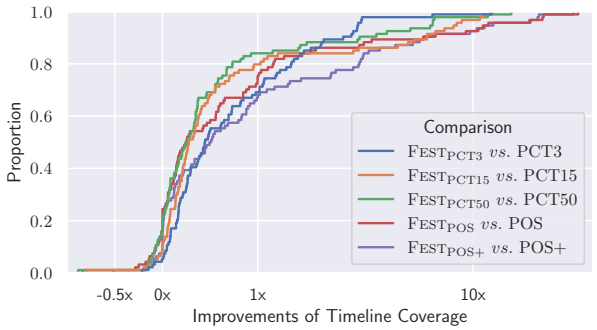

Figure 5: Cumulative distribution function of improvements over the original approach without feedback (e.g., $\text{FEST}_{\text{PCT3}}$ versus PCT3) for different schedule generation algorithms.

5.3 RQ1: Timeline Coverage Improvement

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

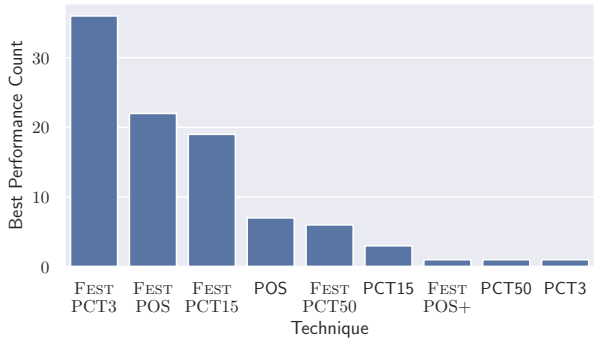

Figure 6: Best performance count: Number of test cases for which each technique achieves the highest timeline coverage.

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 6 illustrates the number of test cases for which each technique achieves the highest timeline coverage. The FEST_{PCT} technique with a bound 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.

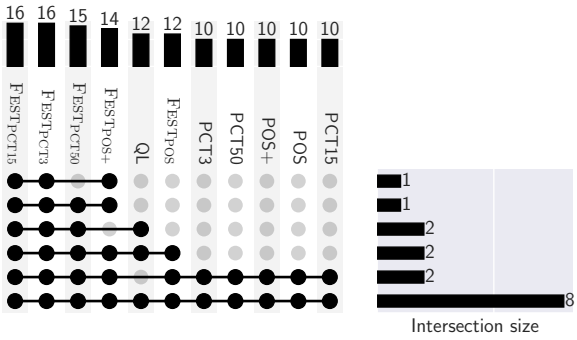

Figure 7: Upset plot for bug counts across different scheduling algorithms. Higher is better.

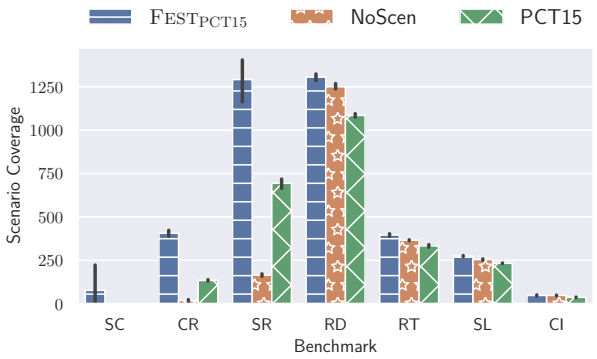

Figure 8: Comparison of timeline coverage that satisfies the specified scenario across different techniques.

5.4 RQ2: Bug Finding Capabilities

Figure 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, FESTPCT3 and FESTPCT15 proved superior, identifying 16 bugs, while PCT3, PCT15, PCT50, POS, and POS+ identified only 10 and QL only identified 12. Notably, FESTPCT3 and FESTPCT15 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.

5.5 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 2.

Figure 8 shows the scenario coverage achieved by different techniques. To save space, we present results from PCT15 and FESTPCT15. Across all benchmarks, FESTPCT either outperforms or matches NoScen, highlighting the value of incorporating scenario feedback into feedback guidance. Moreover, FESTPCT consistently outperforms PCT on all benchmarks. In the elastic storage model with the concurrent split-merge (SC) scenario, FESTPCT identifies up to 39× 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× and 1.7× improvements compared to PCT, respectively. For stream log scenarios (RD, RT, and SL) and memory DB with transition read scenario (CI), the improvement FESTPCT 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 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.

5.6 RQ4: Priority-based Schedule Generation

Figure 9 illustrates the timeline coverage improvement of FESTPCT15 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×. 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

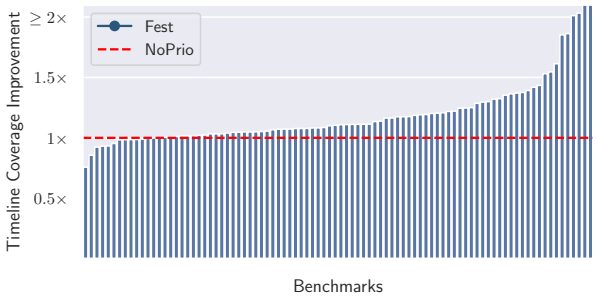

Figure 9: Timeline coverage improvement of FEST$_{PCT15}$ compared to the version without priority-based scheduling (NoPrio). The y-axis is cut off at 2× due to space limitations.

the efficiency of FEST in covering more unique timelines.

6 Related Work

Greybox Fuzzing. Greybox fuzzing [7, 8, 14, 15, 23, 25, 31, 34, 36, 45, 46] strikes a balance between white-box analysis, an expensive symbolic technique with deep introspection of the system under test, and black-box testing, an efficient approach devoid of system-internal knowledge. Starting with an initial corpus of seed inputs, a greybox fuzzer iteratively enhances this corpus by synthesizing new inputs that exhibit desirable characteristics, such as increased line coverage.

Distributed System Testing. Research in distributed system testing can be broadly categorized into approaches with complete control over scheduling (e.g., [26, 32, 33, 43, 54, 56]) and those with partial or no control (e.g., [4, 38, 49]). Many prior works reduce the search space using system semantics through techniques like dynamic partial order reduction [54], dynamic interface reduction [26], and semantic-aware model checking [32]. These approaches effectively prune the search space but still use random exploration for scheduling. FEST is orthogonal to these techniques—it improves the scheduling algorithm itself to maximize behavioral coverage. Combining these reduction approaches with FEST would likely be most effective, as better scheduling would work on an already-pruned search space. In contrast to these approaches, Mallory employs reinforcement learning for heuristic insertion of failures in distributed systems [38]. However, the learning-based approach (similar to QL in our evaluation) cannot integrate with existing bounded-concurrency testing methods, and also, as demonstrated by our results, FEST performs better than QL for coverage.

Distributed System Verification. Distributed system verification has garnered significant attention in both industry and academia, focusing on both their implementation [27, 28, 42, 54] and protocol design [18, 30, 35, 37, 48]. While systematic verification of distributed system correctness remains the gold standard, it has become increasingly impractical for modern systems due to their growing scale and accelerated development cycles [11]. Consequently, practitioners have adopted lightweight formal methods that integrate formal specifications with property-based testing and fuzzing, striking a pragmatic balance between bug detection and correctness assurance [10].

Coverage Measurements for Distributed Systems. Researchers have explored various coverage measurements to assess the unique behaviors explored by testing tools [17, 18, 38, 50]. Similar to abstract Lamport timelines, location pairs leverage the "happens before" relationships [50]. FEST uses the abstract Lamport timeline to measure the behavior coverage. Additionally, the algorithm can be easily extended to support other coverage measurements.

Concurrency Testing. Like distributed system testing, concurrency testing is primarily concerned with evaluating a target program under various thread inter-leavings [13, 40, 44, 51, 52, 53, 55]. RFF leverages read-write relationships to construct abstract schedules, guiding the schedule generation process [53]. PERIOD utilizes the DEADLINE scheduler implemented in Linux kernel, which parallelizes threads within defined periods rather than searching through interleavings [52]. None of the existing work integrates the bounded-concurrency testing algorithms with feedback-guided mutation-based fuzzing, resulting in an inability to explore unique behaviors efficiently. Also, they cannot guide the schedule generation algorithm towards a specific scenario.

7 Conclusion

FEST advances distributed system design testing by introducing feedback-driven adaptive schedule generation. It addresses the limitations of existing randomized search-based bounded concurrency testing techniques by combining adaptive schedule generation with diversity feedback to enhance behavioral exploration. Our approach integrates record-and-replay techniques with parametric input generation, allowing for controlled mutations of schedules while preserving scheduling constraints. The diversity feedback mechanism effectively prioritizes schedules that are more likely to reveal unique behaviors by evaluating the uniqueness of each schedule using abstract Lamport timeline. Additionally, FEST introduces a novel scenario specification language, enabling developers to guide testing toward specific, critical scenarios. Our evaluation demonstrates that FEST significantly improves both behavioral and scenario coverage compared to state-of-the-art algorithms, resulting in the detection of more bugs across a diverse set of distributed system models.

8 Acknowledgments

This work was partially conducted at Amazon as part of an AWS Applied Science internship, and partially supported by the National Science Foundation through grant number CCF-2453432. We thank the anonymous reviewers for their insightful comments and constructive feedback, and Doug Terry and Vyas Sekar for their feedback on earlier drafts.

References

[1] Fire-flyer file system. https://github.com/deepseek-ai/3FS/tree/main/specs.

[2] Complex Event Processing, Streaming SQL and Event Series Analysis for Java. https://github.com/espertechinc/esper.

[3] FlinkCEP - Complex event processing for Flink. https://nightlies.apache.org/flink/flink-docs-master/docs/libs/cep/.

[4] Jepsen: Distributed systems safety research. https://jepsen.io/.

[5] P - Case Studies. https://p-org.github.io/P/casestudies/.

[6] Formal modeling and analysis of distributed (event-driven) systems. https://github.com/p-org/P.

[7] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019. doi: 10.14722/ndss.2019.23371.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGPLAN Not.*, 45(3):167–178, mar 2010. ISSN 0362-1340. doi: 10.1145/1735971.1736040. URL https://doi.org/10.1145/1735971.1736040.

[13] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu.

[14] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018. doi: 10.1109/SP.2018.00046.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, apr 1986. ISSN 0164-0925. doi: 10.1145/5397.5399. URL https://doi.org/10.1145/5397.5399.

[16] Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, and Koen De Bosschere. A taxonomy of execution replay systems. 01 2003.

[17] Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. Industrial-strength controlled concurrency testing for c# programs with coyote. In *TACAS*, April 2023.

[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.

[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.

[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.

[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.

[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.

[23] Zhen Yu Ding and Claire Le Goues. An empirical study of oss-fuzz bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142. IEEE, 2021. doi: 10.1109/MSR52588.2021.00026.

[24] 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.

[25] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594, 2020. doi: 10.5555/3489212.3489357.

[26] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 265–278, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309776. doi: 10.1145/2043556.2043582. URL https://doi.org/10.1145/2043556.2043582.

[27] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815428. URL https://doi.org/10.1145/2815400.2815428.

[28] Nouraldin Jaber, Swen Jacobs, Christopher Wagner, Milind Kulkarni, and Roopsha Samanta. Parameterized verification of systems with global synchronization and guards. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 299–323, Cham, 2020. Springer International Publishing. ISBN 978-3-030-53288-8.

[29] 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.

[30] 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.

[31] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages*, 3 (OOPSLA):1–29, 2019. doi: 10.1145/3360607.

[32] 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.

[33] 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.

[34] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019. doi: 10.5555/3361338.3361473.

[35] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359651. URL https://doi.org/10.1145/3341301.3359651.

[36] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019. doi: 10.1109/TSE.2019.2946563.

[37] Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*, page 190–202, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-53290-1. doi: 10.1007/978-3-030-53291-8_12. URL https://doi.org/10.1007/978-3-030-53291-8_12.

[38] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems, 2023. URL https://doi.org/10.1145/3576915.3623097.

[39] 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.

[40] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 267–280, USA, 2008. USENIX Association.

[41] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 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.

[42] Lingzhi Ouyang, Xudong Sun, Ruize Tang, Yu Huang, Madhav Jivrajani, Xiaoxing Ma, and Tianyin Xu. Multi-grained specifications for distributed system model checking and verification. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 379–395, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3696069. URL https://doi.org/10.1145/3689031.3696069.

[43] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi: 10.1145/3276530. URL https://doi.org/10.1145/3276530.

[44] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360606. URL https://doi.org/10.1145/3360606.

[45] Rohan Padhye, Caroline Lemieux, and Koushik Sen. JQF: Coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA'19, pages 398–401, 2019. ISBN 978-1-4503-6224-5. doi: 10.1145/3293882.3339002. URL http://doi.acm.org/10.1145/3293882.3339002.

[46] 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.

[47] 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.

[48] 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.

[49] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 143–159, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/sun.

[50] Serdar Tasiran, M. Keremoğlu, and Kivanç Muşlu. Location pairs: A test coverage metric for shared-memory concurrent programs. *Empirical Software Engineering*, 17:129–165, 06 2012. doi: 10.1007/s10664-011-9166-8.

[51] 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, page 474–486, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510178. URL https://doi.org/10.1145/3510003.3510178.

[52] 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, page 474–486, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510178. URL https://doi.org/10.1145/3510003.3510178.

[53] Dylan Wolff, Zheng Shi, Gregory J. Duck, Umang Mathur, and Abhik Roychoudhury. Greybox fuzzing for concurrency testing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 482–498, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640389. URL https://doi.org/10.1145/3620665.3640389.

[54] 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 *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*, Boston, MA, April 2009. USENIX Association. URL https://www.usenix.org/conference/nsdi-09/modist-transparent-model-checking-unmodified-distributed-systems.

[55] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 485–502. ACM, 2012. doi: 10.1145/2384616.2384651. URL https://doi.org/10.1145/2384616.2384651.

[56] 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.

[57] 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.

Name	PCT3		PCT15		PCT50		POS		POS+		QL
	Ori	FEST	Ori	FEST	Ori	FEST	Ori	FEST	Ori	FEST	
3FS01	33	61	272	322	375	421	1893	**2384**	45	518	106
3FS02	67	116	337	399	397	485	5677	**7250**	69	873	270
3FS03	32	66	290	362	396	474	4473	**4886**	190	604	127
3FS04	31	73	280	338	416	478	3163	**3521**	165	571	120
3FS05	27	59	253	304	363	424	2808	**2965**	117	452	107
3FS06	26	61	238	314	365	486	**1251**	1232	37	74	59
3FS07	19	55	161	290	452	572	**859**	787	17	37	33
3FS08	27	69	268	342	373	482	**2104**	1967	48	151	66
3FS09	65	118	337	416	386	513	**4677**	4620	244	674	220
3FS10	31	75	282	356	400	539	2182	**2489**	35	191	77
3FS11	35	77	306	371	409	556	2330	**2706**	46	208	85
3FS12	28	51	250	332	361	494	**1837**	1756	54	177	91
Access Management01	1183	**1494**	575	1271	166	302	27	32	17	100	532
Access Management02	42	**68**	5	4	4	4	2	2	2	2	28
Access Management03	1343	1685	2265	**2573**	399	968	45	276	41	132	908
Bucket Storage01	603	473	334	**702**	133	137	120	90	58	53	106
CDN01	138	**367**	37	205	30	112	1	1	1	1	23
CDN02	56	**167**	10	64	10	10	1	1	2	2	1
Concurrency Control01	5610	**8319**	5198	6535	4150	4864	3714	4650	4023	4922	626
Concurrency Control02	19362	25058	21760	26357	20450	**27101**	17419	25521	17709	25590	3169
Concurrency Control03	13588	16085	16408	**18564**	17215	18524	14822	17891	14774	16503	4001
Concurrency Control04	9826	**14515**	6344	12375	2398	6024	2864	9467	3847	10658	937
Concurrency Control05	37897	39027	**45741**	39470	31990	33252	24655	35265	24704	34765	5525
Concurrency Control06	37909	37676	**42682**	39391	40604	36570	35654	37224	35791	36350	4796
Database Storage Engine01	401	**996**	67	523	24	159	13	343	9	227	342
Database Storage Engine02	456	**1018**	142	736	70	290	41	481	30	318	153
Database Storage Engine03	418	**961**	269	807	350	472	140	499	180	358	189
Database Storage Engine04	512	**1218**	138	881	23	407	14	588	10	396	281
Database Storage Engine05	391	**1052**	60	542	26	138	15	339	11	245	200
Database Storage Engine06	404	**1259**	44	551	41	71	26	48	16	27	126
Database Storage Engine07	302	348	349	70	**359**	41	163	26	29	14	7
Database Storage Engine08	405	**1216**	60	521	47	73	30	45	20	31	126
Database Storage Engine09	378	**1187**	41	490	36	56	25	41	15	23	109
Database Storage Engine10	554	**1136**	141	654	75	234	40	403	43	264	314
Database Storage Engine11	390	**953**	62	595	23	157	13	348	9	223	248
Elastic Storage01	737	629	802	**934**	848	787	678	558	603	611	2
Elastic Storage02	6316	**8151**	5832	5923	5494	5820	5907	6945	5307	6242	2
Elastic Storage03	1002	1494	1473	1549	1325	1121	1149	**1789**	903	1175	2
Elastic Storage04	342	1112	390	664	380	635	498	**1271**	470	1088	56
Elastic Storage05	208	646	250	389	278	536	398	**858**	329	807	30
Elastic Storage06	1578	1604	1721	1716	1473	1649	1595	**1824**	1564	1586	2
Elastic Storage07	293	371	273	478	287	498	392	**789**	371	728	2
Elastic Storage08	374	404	391	**413**	374	372	321	332	364	362	2
Elastic Storage09	183	215	199	**227**	203	218	143	163	163	183	2
Elastic Storage10	76	**92**	42	65	17	14	16	25	13	15	3
Elastic Storage11	65	**68**	49	61	31	31	32	32	32	32	2
Elastic Storage12	5679	5941	5561	5846	5542	5811	**6053**	5914	5159	5199	2
Elastic Storage13	68	**74**	47	58	30	29	30	29	29	28	2
Elastic Storage14	324	**405**	**405**	385	278	294	258	296	180	184	2
Elastic Storage15	367	**604**	363	413	263	317	281	451	226	330	2
Elastic Storage16	727	830	792	**849**	679	726	533	564	547	574	2
Elastic Storage17	330	**385**	329	355	241	229	238	266	138	141	2
Elastic Storage18	532	1073	519	881	551	629	692	**1409**	561	1175	2
Elastic Storage19	207	223	219	**225**	177	205	125	161	158	211	2
Elastic Storage20	1114	**1920**	1226	1829	1146	1810	885	1633	755	1447	2
Elastic Storage21	1585	2282	1831	2523	1613	2062	1302	**2600**	1013	1820	2
Elastic Storage22	265	310	269	**311**	218	270	160	227	197	294	2
Elastic Storage23	300	508	382	459	319	590	468	**1014**	378	626	2
Elastic Storage24	275	**314**	219	280	172	174	126	133	116	130	2
Elastic Storage25	1094	1890	1272	1836	1205	1454	927	**1952**	737	1397	2
Elastic Storage26	1690	**2862**	1741	2573	1535	2112	1262	2087	1070	2163	2
Elastic Storage27	57	**72**	43	43	24	24	24	24	18	18	3
Elastic Storage28	454	565	378	604	368	572	423	**922**	348	813	2
Elastic Storage29	2278	2088	1759	2367	1976	1678	2198	**2478**	1825	2071	2

Continued on next page

Name	PCT3		PCT15		PCT50		POS		POS+		QL
	Ori	FEST	Ori	FEST	Ori	FEST	Ori	FEST	Ori	FEST	
Elastic Storage30	1509	1603	1427	1806	1656	1439	1801	**2076**	1447	1701	2
Elastic Storage31	698	1409	806	1078	1231	1131	1645	**2061**	1308	1691	2
Elastic Storage32	551	1090	726	885	999	985	1389	**1703**	1195	1597	2
Financial Service01	268	**319**	264	263	251	229	142	132	166	155	5
Financial Service02	58	**63**	55	54	36	39	7	5	33	33	1
Financial Service03	**233**	209	216	202	199	204	142	127	151	132	3
Financial Service04	177	**245**	173	186	170	161	132	114	114	96	4
FreeRTOS01	2067	**2896**	1867	2619	997	1344	1111	2221	1134	2131	540
FreeRTOS02	2132	**2889**	1866	2608	1009	1378	1138	2170	1124	2053	549
FreeRTOS03	2061	**2814**	1894	2541	1000	1464	1113	2223	1126	2064	526
FreeRTOS04	2064	**2940**	1870	2561	997	1336	1109	2196	1125	2113	570
German01	227	229	229	**230**	60	60	209	209	209	209	87
Leader Election01	3725	4839	10301	14557	27919	25184	**41932**	35925	34706	28720	31645
MemoryDB01	79	**92**	81	**92**	82	87	44	44	16	16	6
Message Broker01	21	56	34	**69**	60	65	12	12	8	9	20
Network Dynamic Scaling01	949	1526	966	1461	929	**1556**	939	1491	427	617	4
Network Routing01	1206	2494	1619	3338	1583	**3413**	973	1544	378	1134	185
Network Routing02	1217	2361	1389	2720	1474	**3605**	233	2888	318	896	231
OSR01	1170	1694	1224	2026	125	853	1718	2679	2465	**2994**	162
Paxos01	93	**1284**	92	1258	95	1253	94	1249	89	1258	2
Paxos02	2541	8201	2610	**8237**	2562	8149	2556	8108	2462	8126	63
Paxos03	571	3880	553	3897	569	**3955**	555	3912	557	3817	1
Paxos04	19221	20859	19213	20812	19315	**20906**	19087	20858	18989	20726	26
Stream Log01	1009	1454	1697	1981	733	1192	1124	**2110**	1124	1556	311
TwoPhaseCommit01	2136	3039	2763	**3780**	1240	1808	723	1645	1566	2367	788
TwoPhaseCommit02	1979	2719	2355	**3237**	1097	1833	605	1579	1357	2187	774
TwoPhaseCommit03	1507	2263	1763	**2595**	814	1289	341	841	1164	1772	604
TwoPhaseCommit04	1253	1525	1548	**1626**	754	980	342	367	372	374	552
TwoPhaseCommit05	1360	1591	1607	**1712**	821	1015	372	393	398	401	570
TwoPhaseCommit06	1108	1287	1218	**1319**	624	854	262	305	339	350	471

Table 3: Timeline coverage for different models and techniques each row shows a distributed system implemented in P.

Name	PCT3			PCT15			PCT50			POS			POS+			QL
	Ori	NoScen	FEST	Ori	NoScen	FEST	Ori	NoScen	FEST	Ori	NoScen	FEST	Ori	NoScen	FEST	
CI	38	48	**53**	36	47	47	38	49	49	32	34	34	12	12	12	1
CR	40	20	77	127	18	348	90	42	659	13	24	2625	13	32	**3633**	3
RD	617	943	976	1083	1248	1301	566	792	779	900	**1401**	1387	851	1026	1061	267
RT	79	159	176	333	365	395	289	324	316	387	455	**484**	354	378	394	128
SC	0	0	5	0	0	**39**	0	0	2	0	0	0	0	0	0	0
SL	96	152	164	232	254	273	193	223	223	275	327	**350**	255	271	287	74
SR	209	106	248	656	149	1140	964	339	2526	216	291	4315	161	283	**5583**	400

Table 4: Scenario coverage for different models and techniques. Each row shows a distributed system and its scenario coverage.

A Timeline Coverage

Table 3 shows the timeline coverage achieved by each technique. Each cell in the table displays the average unique timelines explored by each technique over five iterations. For each technique, except QL, we compare the timeline coverage achieved by the original random sampling method (Ori) and the feedback-guided schedule generation (FEST). Shaded cells indicate instances where FEST achieves higher timeline coverage than its random sampling counterpart. Additionally, cells with **bold red** values indicates that the corresponding technique achieves the highest timeline coverage.

Table 4 shows the scenario coverage achieved by each technique. Similar to timeline coverage, each cell in the table displays the average unique timelines that satisfy the input scenario explored by each technique over five iterations. For each technique, except QL, we compare the timeline coverage achieved by the original random sampling method (Ori), the feedback-guided schedule generation (FEST), and the feedback-guided schedule generation without scenario feedback (NoScen).