

Model-Guided Fuzzing for Apache Accord

Evaluating Predicate-Based Coverage as a Practical
Alternative to TLA+

Stefan Stoicescu



Model-Guided Fuzzing for Apache Accord

Evaluating Predicate-Based Coverage as a
Practical Alternative to TLA+

by

Stefan Stoicescu

Chair/Supervisor:	Dr. Burcu Kulahcioglu Ozkan
Daily Supervisor:	Berkay Gulcan
Committee Member:	Dr. Jérémie Decouchant
Committee Member:	Dr. Asterios Katsifodimos
External Committee Member:	Dr. Alexey Gotsman, IMDEA Software Institute
External Committee Member:	Benedict Elliott Smith, Apple
Project Duration:	December 2025 – August 2026
Faculty:	Electrical Engineering, Mathematics and Computer Science, Delft

Cover: Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA
under CC BY-NC 2.0 (Modified)

Style: TU Delft Report Style, with modifications by Daan Zwaneveld

Acknowledgements

I would like to thank my supervisors, Burcu Ozkan and Berkay Gulcan, who met with me on an almost weekly basis over the course of the project. Their feedback shaped the approach and direction of the thesis, and the design of the predicate abstraction. I am grateful for their support and consistent involvement.

Several people outside of TU Delft have also been generous with their time. Benedict Elliott Smith helped me understand Accord's implementation and gave me a clear sense of whether the choices I was making in the harness were sensible. Alexey Gotsman and Fedor Ryabinin kept me informed of developments in the algorithm and its specification, and Alexandre Siret wrote the TLA+ specification on which the model-guided part of this work depends.

Abstract

Distributed consensus protocols are the backbone of modern cloud infrastructure, ensuring consistency across replicated state machines. Leaderless consensus algorithms, such as Apache Accord, achieve optimal latency (1-RTT) by effectively distributing the ordering load across the cluster; however, this comes at the cost of significant algorithmic complexity during fault recovery. When a coordinating node crashes, surviving replicas must reconstruct dependency and ordering information by combining partial histories. A recent report by Ryabinin et al. demonstrates that non-trivial linearizability bugs can occur in Accord’s recovery logic under very specific network and node failure conditions.

Finding such flaws through trace-guided fuzzing is infeasible because the number of message interleavings makes almost every execution appear distinct. We adapt ModelFuzz, which uses abstract TLA+ states as a coverage signal, by extending Accord’s testing framework with controlled message delivery, crash injection, and a mapping from implementation executions to the formal model. We also introduce predicate coverage, a lighter-weight alternative derived from protocol stages, quorum progress, and crashed replicas. ModelFuzz achieves the highest TLA+ state coverage, while the most detailed predicate configuration can outperform unguided baselines but remains significantly behind ModelFuzz. Controlled exploration also exposes a reproducible internal correctness failure in Accord.

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
1.1 Research Questions	2
1.2 Contributions	2
1.3 Thesis Structure	2
2 Background	4
2.1 Primer on Leaderless Consensus	4
2.2 Apache Accord: System Model	4
2.3 Apache Accord: Steady-State Transaction Flow	5
2.4 Apache Accord: Recovery Branches and Quorums	5
2.5 Apache Accord: Published Linearizability Violation	6
3 Methodology	9
3.1 ModelFuzz	9
3.2 Coverage by Predicates	10
3.2.1 Motivation	10
3.2.2 Motivating Example	11
3.2.3 Design	11
3.2.4 Predicate State Machine	13
3.2.5 Evaluated Configurations	14
4 Implementation	16
4.1 BurnTest	16
4.2 Scheduling	17
4.3 Mapping Executions to Abstract States	18
4.3.1 Action-Message Discrepancy	18
4.3.2 One Message, Several Actions	19
4.3.3 Actions Without a Message	19
4.3.4 Controlled-Mode Specification Changes	19
4.3.5 Recovering Absent Correlations	20
4.3.6 Identities and Timestamps	20
4.4 Guidance	20
5 Experimental Evaluation	22
5.1 Experimental Setup	22
5.1.1 Protocol Configurations and Workload	22
5.1.2 Execution Environments	22
5.1.3 Fuzzer Configuration	22
5.1.4 Evaluation Metrics	23
5.2 Fault Discovery and Characterization	23
5.3 Fault-Finding Effectiveness	23
5.4 Abstract-State Coverage	24
5.4.1 Predicate Abstraction Variants	24
5.5 Five- and Nine-Replica Configurations	27
5.6 Discussion	29
5.6.1 Summary of Findings	29
5.6.2 Crash-State Limitation	29

5.6.3	Mapping Fidelity and State-Space Growth	29
5.6.4	Fault-Finding Results	30
5.6.5	Limitations	30
6	Related Work	32
6.1	Systematic Schedule Exploration	32
6.2	Probabilistic Schedule Sampling	32
6.3	Semantic and Feedback-Guided Exploration	33
7	Future Work	35
8	Conclusion	36
	References	37
A	Trace for Iteration 58,372	39

1

Introduction

Distributed services underpin much of the modern Internet, from the storage systems behind large online applications to the coordination infrastructure on which these applications depend [6, 7, 5, 11, 13]. These services are expected to remain dependable even though the machines on which they run may crash and the networks between them may delay or lose messages [2]. Redundancy is the simplest solution for tolerating these failures, but replication is only useful if the copies continue to behave as one coherent service. State-machine replication (SMR) provides the fundamental abstraction for achieving this: a service is expressed as a deterministic state machine, replicated across several nodes, and driven by an agreed sequence of commands. Replicas that begin in the same state and execute the same commands in the same order produce the same results, allowing clients to observe a single reliable service despite individual failures [25, 20]. Consensus and SMR are therefore basic building blocks of modern distributed infrastructure.

The agreement required by SMR is commonly implemented by routing commands through a single leader, as in Multi-Paxos, and Raft [19, 14, 20]. Having a central point of coordination makes ordering straightforward, but it can become a throughput bottleneck, add latency for distant clients, and halt progress while a failed leader is replaced. Leaderless protocols such as Egalitarian Paxos (EPaxos) [17] instead allow any replica to coordinate a command, distributing both the load and the opportunity to make progress across the cluster.

Apache Accord applies this leaderless approach to transactions in partitioned and geo-distributed databases. It orders transactions using logical timestamps and can commit on its Fast Path in one wide-area round trip when replicas agree, while retaining fallback paths for conflicts and failures [27, 26]. These latency and availability advantages come with a drawback, namely that no single replica is certain to observe the complete history of a transaction. If a coordinator fails, another replica must reconstruct the transaction's state from a quorum whose members may have observed different phases, timestamps, and conflicting transactions. Recovery must preserve any decision that may already have been made, even when that decision was only partially propagated.

This recovery logic is difficult to reason about and test. Ryabinin et al. describe an allowed execution in which a coordinator crashes after executing a transaction but before propagating the decision, after which two recoveries produce an ordering incompatible with the earlier execution [23]. Reproducing the violation requires a narrow combination of message delays, a Slow Path decision, a crash, and carefully selected recovery quorums, making it unlikely to arise under uniform random scheduling.

Schedule exploration offers a way to search for such executions [29, 15, 22, 21]. Coverage-guided exploration additionally needs a useful metric of progress. Conventional code coverage is too coarse for this setting because many different message orderings execute the same handlers. On the other hand, treating every delivery trace as distinct means that harmless reorderings produce an enormous number of apparently new executions, and therefore novelty becomes almost guaranteed and provides little guidance. A coverage signal for distributed protocols must lie between these two extremes. It should

identify schedules that reach meaningfully different protocol states while collapsing differences that do not affect protocol behaviour.

ModelFuzz [10] addresses this problem by mapping a concrete execution to actions in a TLA+ specification and using the resulting abstract states as coverage. Schedules that reach previously unseen model states are retained and mutated, directing exploration toward new protocol behaviour rather than new instruction paths or raw traces. This is a compelling fit for Accord because an abstract model naturally records the phases and quorum decisions on which its recovery branches depend.

Applying this idea to a production implementation still requires translating concrete executions into model actions. Because the implementation and specification encode the protocol at different levels of detail and evolve independently, constructing and maintaining this mapping requires protocol-specific engineering. The model provides an effective abstraction, but obtaining its states from real executions may therefore be costly.

This thesis investigates whether useful guidance can be obtained with a lighter-weight abstraction. The proposed predicate-based coverage records how far each transaction has progressed through Accord’s protocol phases, how many replicas have received a request or returned a reply, whether those counts cross a quorum, and which replicas have crashed. These predicates are derived directly from the execution trace and message vocabulary. They preserve distinctions that matter to schedule exploration, such as whether a transaction has reached only a subset of replicas or has crossed a quorum, without replaying the execution through a formal specification. Accord serves both as a demanding case study and as a setting in which predicate coverage can be compared against TLA+ state coverage using the same executions.

1.1. Research Questions

This thesis centres on the following research questions:

- RQ1:** Can controlled schedule exploration expose a correctness violation in Accord, and what execution conditions produce it?
- RQ2:** How do TLA+ model guidance, predicate-based guidance, and unguided random exploration compare in their ability to find bugs?
- RQ3:** How do TLA+ model guidance and predicate-based guidance compare with unguided random exploration in the number of distinct abstract states they discover and in how their coverage evolves over time?

1.2. Contributions

To answer these questions, we make the following contributions:

- We adapt Accord’s deterministic simulation-based testing environment, BurnTest, into a schedule-exploration harness that can record executions, control network-message delivery, replay mutated schedules, and inject crash-stop failures without replacing the production protocol logic.
- We implement model-guided fuzzing for Accord by translating concrete deliveries into actions of Accord’s TLA+ specification and document the points at which the implementation and model do not correspond directly.
- We introduce a family of predicate-based coverage abstractions derived from transaction phases, request and acknowledgement counts, quorum thresholds, and failures. These abstractions avoid dependence on a formal model while retaining protocol-relevant progress.
- We compare two unguided baselines, TLA+ model-guided fuzzing, and predicate-guided fuzzing using abstract-state coverage and their ability to expose and reproduce faulty Accord executions.

1.3. Thesis Structure

Chapter 2 introduces leaderless consensus, Accord’s steady-state and recovery protocols, and the known faulty execution that motivates the work. Chapter 3 explains model-guided fuzzing and develops the predicate-based coverage abstraction. Chapter 4 describes the BurnTest scheduler, the mapping between

concrete executions and the TLA+ model, and the guidance loop. Chapter 5 evaluates the guidance strategies and unguided baselines and discusses their limitations. The final chapters place the results in the context of related and future work and summarize the conclusions of the thesis.

2

Background

This chapter provides the protocol background needed to motivate and interpret the testing approach developed in this thesis. It first places Accord in the context of leaderless consensus, then describes its system model, steady-state transaction flow, and recovery branches. The chapter concludes with a published faulty execution that illustrates how individually valid message delays, crashes, and quorum choices can combine into a correctness violation. This execution is included as motivation and as a concrete example of the protocol information that useful schedule guidance should preserve, rather than as the failure targeted by the experiments.

2.1. Primer on Leaderless Consensus

Consensus protocols such as Multi-Paxos achieve fault tolerance by routing every command through a single elected leader [14, 20]. While funnelling every command through one node makes ordering straightforward, this approach introduces significant latency for geographically distant clients and creates a single point of failure whenever the leader becomes unavailable.

To address the single-leader bottleneck, Egalitarian Paxos (EPaxos) [17] introduces a leaderless architecture where any replica can act as a command leader, allowing for higher throughput and better load distribution. EPaxos orders commands by building a dependency graph, where every command tracks which other commands interfere with it. If a command's dependencies are consistent across a fast quorum (supermajority), it commits on the Fast Path in just 1-RTT. However, if replicas report conflicting dependencies, the leader must fall back to the Slow Path (2-RTT) to explicitly finalise the order, adding complexity to conflict resolution.

While EPaxos solved the performance bottleneck, its recovery logic introduced a linearizability bug [28]. During recovery, two leaders attempting to recover conflicting commands could enter a recursive cycle where they repeatedly overwrite each other's proposed dependencies [24]. This illustrates a broader pattern, namely that in leaderless protocols recovery tends to be the most algorithmically complex and fragile component. Accord, the subject of this thesis, follows in the same leaderless lineage but replaces dependency graphs with timestamp-based ordering, and faces its own recovery challenges as a result.

2.2. Apache Accord: System Model

Accord operates on n processes, also called replicas, communicating over an asynchronous network in which messages may be arbitrarily delayed or lost. Up to f of these processes may fail by crashing, the standard failure assumption for consensus protocols [26, 27]. Clients submit transactions, and any process may subsequently act as the coordinator of a transaction it receives. Two transactions are said to conflict if they access overlapping keys, and the central correctness requirement is that all replicas execute conflicting transactions in the same order.

In this chapter, we use the basic, single-partition version of Accord, in which every replica holds a copy of the full data set and participates in every transaction. Both the original whitepaper [27] and the

formal protocol description [26, 27] introduce the algorithm in this setting, and it is sufficient to expose the recovery dynamics that this thesis investigates. The multi-shard generalization, where the key space is split within nodes and only the shards touched by a transaction participate in ordering it, follows naturally from the single-partition case.

A transaction can complete on the Fast Path in a single wide-area round trip (1-RTT) when no more than e replicas have failed, where $e \leq \frac{n-f+1}{2}$, the optimal bound for leaderless protocols [26, 27]. In this case the coordinator reaches a fast quorum of $n - e$ replicas. When this condition is not met, Accord triggers a fallback to the Slow Path, requiring one or two additional round trips on a smaller classical quorum of $n - f$ replicas. For the cluster sizes typical in practice ($n \leq 6$), the bound permits setting $e = f$, so fast quorums coincide with simple majorities and the Fast Path remains available under any minority of failures [26, 27].

2.3. Apache Accord: Steady-State Transaction Flow

Each transaction in Accord is identified by a unique ID and progresses through a sequence of phases at every replica that processes it: initial, preaccepted, accepted, committed, and stable [26, 27]. Ordering is determined by timestamps rather than by an explicit dependency graph. When a coordinator receives a transaction tx from a client, it assigns tx an initial timestamp t_0 . This timestamp is derived from the coordinator's local clock and process identifier to break ties between concurrent transactions [26, 27]. It then broadcasts a PreAccept message carrying tx and t_0 to all replicas. Each replica inspects its local state for transactions that conflict with tx , computes a candidate timestamp at least as large as t_0 and as large as any conflicting transaction it has already seen, and replies with a PreAcceptOK message carrying this proposed timestamp together with the set of conflicting transactions it knows about. At this point the replica is said to have preaccepted tx .

How the protocol proceeds depends on the agreement among the responses collected. If a fast quorum of $n - e$ replicas all return the original timestamp t_0 unchanged, no replica in the quorum has seen a conflicting transaction with a higher timestamp, and tx can safely commit with t_0 as its final timestamp. The coordinator then broadcasts Commit and Stable messages, and the transaction is executed once its dependencies have been resolved. This is the Fast Path, completing in 1-RTT from the coordinator to the quorum.

When the responses do not agree, either because some replicas proposed higher timestamps or because only a smaller classical quorum of $n - f$ replicas could be reached, the protocol must explicitly negotiate the final timestamp and dependency set. The coordinator takes the maximum of the proposed timestamps and the union of the reported dependency sets, and sends an Accept message asking the quorum to lock in these values. Each replica that processes an Accept records the timestamp and dependencies, transitions the transaction to the accepted phase, and replies with an AcceptOK. Once a quorum of AcceptOKs is received, the coordinator broadcasts Commit. Replicas that receive Commit reply with CommitOK, and only after a quorum of these acknowledgements is received does the coordinator broadcast Stable, marking the transaction as eligible for execution. This is known as the Slow Path, costing two additional round trips relative to the Fast Path.

The two paths leave the recovery protocol with very different starting points. The Slow Path's Accept and Commit rounds give a durable timestamp and dependency set to build on, whereas the Fast Path leaves only the preaccepted state of surviving replicas. Most of the recovery protocol's complexity lives in reconstructing the latter.

2.4. Apache Accord: Recovery Branches and Quorums

If the coordinator of a transaction id is suspected to have failed, any other replica may take over by initiating recovery [26, 27]. To do so, it picks a ballot b higher than any previously used for id and broadcasts a Recover message carrying b . Each replica that receives Recover replies with a RecoverOK message containing its full local view of the transaction: the phase, the timestamp, the dependency set, and the highest ballot $abal$ at which it has accepted id [26, 27]. Recovery proceeds once RecoverOK responses have arrived from a quorum of $n - f$ replicas.

The recovering replica needs to commit id with values consistent with any decision the failed coordinator

may already have reached. Since that decision could have been made on either the Fast Path or the Slow Path, the recovery process branches according to what the responses reveal. If any replica in the quorum reports id as stable or committed, the decision has clearly been made and is simply rebroadcast. If any replica reports id as accepted but no further, a Slow Path agreement on the timestamp and dependency set already exists, and Accept is rerun with the same values to complete the transaction.

The remaining case, namely the Fast Path, is the more complex one. If every replica in the recovery quorum reports id as only preaccepted, it cannot yet be determined whether the failed coordinator committed id on the Fast Path or never reached agreement at all. To distinguish these, the protocol checks whether a sub-quorum of $|Q| - e$ replicas preaccepted id with its initial timestamp t_0 unchanged [26, 27]. If fewer did, the Fast Path was demonstrably impossible and recovery aborts id by replacing its payload with a no-op. If at least that many did, a Fast Path commitment cannot be ruled out, and t_0 must be conservatively proposed as the final timestamp, ensuring it agrees with whatever the failed coordinator may have decided.

Proposing t_0 in the Fast Path recovery case is only safe if no conflicting transaction has already been committed in a way that contradicts it. Accord captures this through the notion of a superseding transaction. Formally, a conflicting transaction id' supersedes id if some replica has committed or accepted id' with a timestamp greater than t_0 and id does not appear in $dep[id']$ [26, 27]. If such an id' exists, recovering id with t_0 would create an execution order inconsistent with the one id' has already locked in, and recovery aborts id with a no-op rather than risk a violation.

2.5. Apache Accord: Published Linearizability Violation

Ryabinin et al. [23] report a possible execution of Accord's recovery protocol that violates strict linearizability. The bug arises from the interaction of two recovery rounds operating on different quorums after a coordinator crash. We describe this scenario below, since it both grounds the experimental setup of later chapters and suggests why purely random testing may struggle to find such bugs in practice.

The published violation affected an earlier version of Accord and has since been fixed in the implementation evaluated here. Reproducing it is therefore not an objective of the experiments, which do not seed or replay the published schedule. The failure reported in Chapter 5 was discovered independently and is distinct from this violation.

In this thesis, the execution below serves three purposes. First, it motivates testing Accord's recovery logic by demonstrating that individually valid message delays, failures, and quorum choices can combine into a strict linearizability violation. Second, it provides a concrete example of the kind of highly constrained schedule that the test harness must be capable of constructing, involving partial propagation, a narrowly timed crash, and several overlapping recovery quorums. Finally, it identifies the protocol-level information that useful guidance should preserve, particularly transaction phases, quorum progress, failures, and which replicas have observed a transaction.

The problematic cluster consists of five replicas $p_1, \dots, p_5$ running single-partition Accord with a configuration of $n = 5$, $f = 2$, and $e = 2$. Two conflicting transactions are submitted concurrently: γ by p_1 and τ by p_5 , with initial timestamps $t_{0\gamma}$ and $t_{0\tau}$ such that $t_{0\tau} < t_{0\gamma}$. The run involves three recovery quorums, denoted $Q_1 = \{p_1, p_2, p_3\}$, $Q_2 = \{p_2, p_3, p_4\}$, and $Q_3 = \{p_3, p_4, p_5\}$. These quorums overlap pairwise, but p_1 is absent from Q_2 and Q_3 .

Figure 2.1 shows the steady-state phase. Transaction τ reaches only p_4 and p_5 at PreAccept, so it remains preaccepted at those two replicas. Transaction γ proceeds on the Slow Path with quorum Q_1 . Replica p_1 collects PreAcceptOKs from Q_1 with empty dependencies, sends Accept and then Commit, and executes γ locally. Importantly, p_1 then crashes before its Commit messages reach p_2 and p_3 , leaving those replicas with γ in the accepted phase but no knowledge that γ has been executed.

Figure 2.2 shows the first recovery. With p_1 unresponsive, p_2 initiates recovery of γ using quorum $Q_2 = \{p_2, p_3, p_4\}$. Replicas p_2 and p_3 report γ as accepted with empty dependencies, but p_4 has not seen γ before, so on receiving Recover it preaccepts γ from scratch and computes its dependencies against its local state, where τ already exists as preaccepted. The coordinator p_2 takes the union of the reported dependency sets, which now includes τ , and commits γ with $dep_\gamma = \{\tau\}$. The cluster's record of γ now

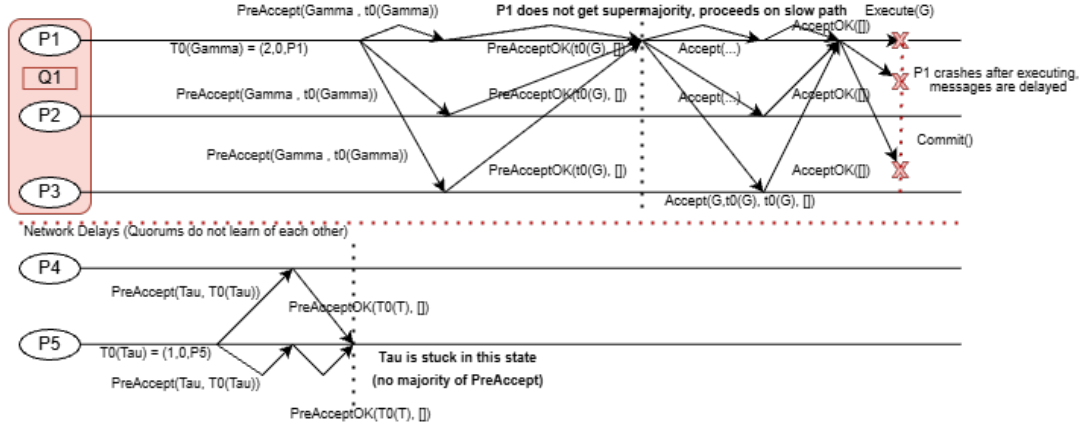


Figure 2.1: The setup phase of the linearizability violation. The coordinator p_1 commits γ on the Slow Path with empty dependencies and executes it locally, then crashes before its Commit messages reach the rest of Q_1 . Meanwhile, τ remains in the preaccepted phase on p_4 and p_5 .

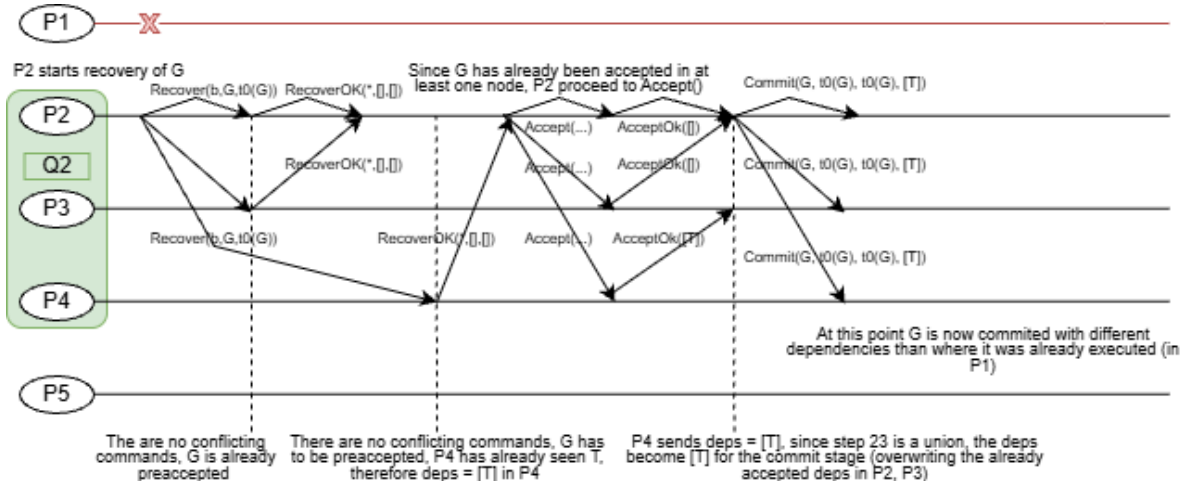


Figure 2.2: Recovery of γ via quorum Q_2 . Replica p_4 , which had preaccepted τ but never seen γ , computes $dep_\gamma = \{\tau\}$ during recovery. The coordinator's union over RecoverOK responses folds τ into γ 's dependency set, overwriting the empty set that γ had actually executed with at p_1 .

disagrees with the version p_1 actually executed.

Figure 2.3 shows the second recovery and the linearizability violation itself. Replica p_4 initiates recovery of τ using quorum $Q_3 = \{p_3, p_4, p_5\}$. No member of Q_3 has committed γ in a way that would identify it as superseding τ , so the superseding-transactions check passes trivially and the protocol commits τ with its initial timestamp $t_{0\tau}$. At this point τ is committed with a timestamp lower than γ 's, meaning the execution order at every surviving replica places τ before γ . But p_1 already executed γ before crashing, with no knowledge of τ . The two execution orders are inconsistent, and strict linearizability is violated.

Reproducing this bug requires a specific combination of events: a Slow Path commit, a coordinator crash after local execution but before commit propagation, a recovery quorum that includes a replica unaware of the executed transaction, and a second recovery quorum that excludes every replica that learned of the first recovery's outcome. Each of these conditions is individually plausible under asynchrony and crash-stop faults, but their occurrence is rare with uniform random message scheduling.

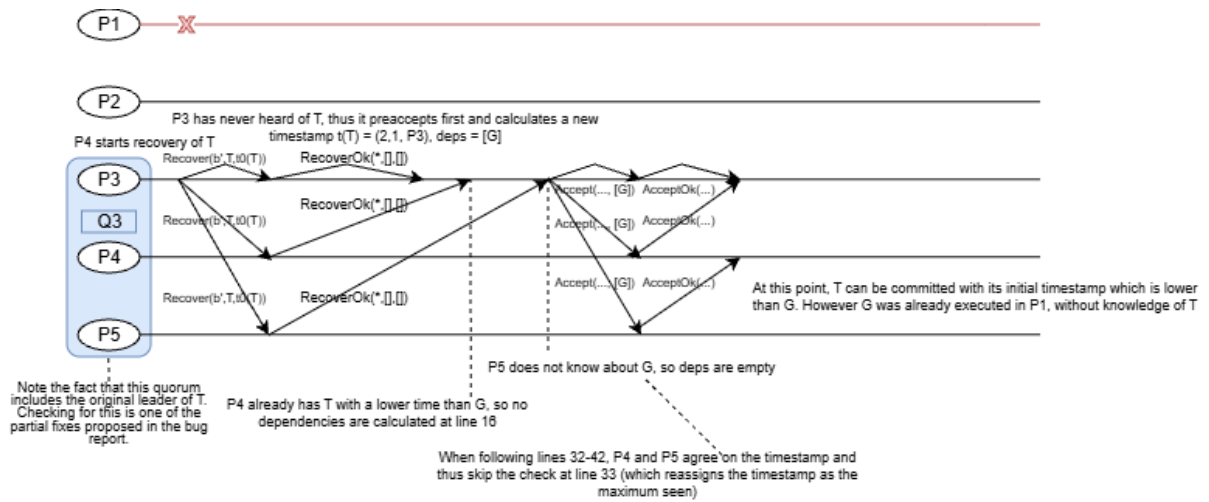


Figure 2.3: Recovery of τ via quorum Q_3 . Because Q_3 excludes every replica that committed γ during the previous recovery, no member of the quorum can identify γ as a superseding transaction. Transaction τ commits with its original timestamp $t_{0\tau} < t_{0\gamma}$, contradicting the execution order at p_1 .

3

Methodology

Coverage-guided fuzzing relies on a notion of coverage to decide whether an execution is worth exploring further. In non-distributed applications, the feedback signal is typically given by code, line, or branch coverage. However, none of these metrics translates well to distributed systems, where the interesting behaviour is determined less by which lines of code run than by the order in which concurrent messages are executed. A single set of code paths can be exercised by many orderings, so a coverage signal for this setting must capture something more meaningful about the ordering of events.

Two natural candidates both fall short. Structural code coverage ignores message orderings entirely, so once a certain handler or piece of code has been executed, no further reorderings register as new coverage, and schedules that distinguish one interleaving from another are never rewarded. Trace coverage sits at the opposite extreme. Treating each distinct sequence of message deliveries as a separate behaviour makes almost every random schedule appear novel, since the number of possible interleavings in Accord is enormous. As a result, the guidance signal degenerates into noise that offers no more direction than random exploration.

An effective signal has to sit between these extremes, coarse enough that insignificant reorderings are not counted as new behaviour, yet fine enough to separate genuinely different protocol states. This is the gap that model-guided fuzzing aims to fill.

3.1. ModelFuzz

ModelFuzz [10] fills the aforementioned gap by measuring coverage over the states of an abstract formal model of the protocol under test. Rather than asking whether an execution reached new code or produced a new message ordering, it asks whether the execution drove the protocol through a combination of states that the abstract model has not previously seen. A TLA+ model is coarse enough that reorderings with no effect on the protocol's logic collapse onto the same state, and fine enough that meaningfully different behaviour produces distinct states. This abstraction is both the source of the method's effectiveness and its main practical cost. The usefulness of the coverage signal depends on how closely the formal model represents the behaviour of the implementation under test. If the implementation evolves while the model does not, relevant protocol branches or optimisations may have no meaningful counterpart in the model, and executions that differ in implementation-relevant ways may become indistinguishable as coverage. Preserving the quality of the signal therefore requires the model and implementation to remain sufficiently aligned as the system evolves.

To make message deliveries controllable, ModelFuzz intercepts every in-flight message and places it in the first-in-first-out (FIFO) buffer associated with its sender-receiver pair. We denote such a buffer by $buff_i$. A test case is then an ordered event schedule whose entries have the form $\langle buff_i : a_i \rangle$, where a_i is either *deliver*, *crash*, or *restart* [10]. A *deliver* event releases the oldest message waiting in the named buffer, while *crash* and *restart* act on its receiving process. Referring to a buffer rather than to a specific message keeps the schedule feasible when its events are later reordered, since the concrete messages waiting in a buffer may differ from one run to the next. Section 4.2 explains how our

implementation reconstructs this per-link buffer abstraction from the simulator’s single global queue.

Coverage is measured by replaying a completed execution against a TLA+ model [10]. Once a schedule has run on the real system, the events it produced are passed through an event mapper, a component supplied by the developer that translates each implementation-level event into a corresponding action of the model. The mapping is described as largely syntactic, matching the type and fields of a delivered message to the corresponding model action, and does not require a deep understanding of the implementation. Replaying the resulting actions on the model produces the sequence of states the execution visits, and this set of visited states is taken as the coverage of the schedule [10].

A delivery does not correspond to a single uniform action. Each message is mapped according to its type and contents to the specific action that handles it, and process crashes and restarts map to their own model actions, so the vocabulary presented to the model is considerably larger than the three kinds of scheduling event [10]. This is where the maintenance cost noted earlier is paid. The mapper has to account for every message type the implementation can emit, and a system that continues to evolve accumulates optimisations and message types that have no counterpart in the abstract model. Each addition must be folded into the mapper by hand or absorbed by extending the model itself, and any unmapped behaviour simply does not contribute to coverage.

These pieces assemble into a coverage-guided loop. Exploration begins from a corpus of randomly generated schedules, each of which is run and assigned its abstract-state coverage. A schedule that reaches a model state not seen in any previous run is considered interesting and is selected for mutation, while one that reveals nothing new is discarded. Mutating an interesting schedule applies a small change intended to produce a similar execution that nonetheless reaches new behaviour. Because a schedule refers to buffers rather than to concrete messages, these mutations remain feasible under reordering. ModelFuzz defines three mutations: swapping the order of two buffers in the schedule, swapping which processes are crashed, and changing the number of messages a delivery releases from a buffer. The loop repeats within a fixed test budget, expanding the corpus with mutated schedules that extend coverage.

The feedback signal is novelty in the abstract TLA+ model rather than novelty in executed code [10], and it is this signal that the remainder of this chapter reconsiders.

3.2. Coverage by Predicates

This section introduces the central proposal of this thesis. It uses predicates derived directly from an implementation execution to define coverage. Rather than replaying the execution through a formal specification, the proposed abstraction summarises each transaction in terms of its protocol phase, its progress toward quorum thresholds, and the failures observed during the execution. The resulting summaries serve as abstract coverage states that can determine which schedules are retained and mutated. The following subsections motivate this alternative, illustrate the protocol information it preserves, and define how its states are constructed.

3.2.1. Motivation

The abstract model states used by ModelFuzz provide a strong coverage metric when the model closely represents the implementation. A formal specification is designed to retain variables that determine protocol behaviour while discarding lower-level details, so novelty over its states captures meaningful differences between executions. The practical difficulty is obtaining those states from a production implementation. The specification and implementation may evolve independently or encode the protocol at different levels of abstraction, making the translation from implementation events to model actions an engineering artefact that must itself be constructed and maintained. Section 4.3 describes these difficulties for Accord, where the resulting correspondence is neither small nor straightforward.

Predicate-based coverage seeks a lighter-weight source of semantic feedback. It records how far each transaction has progressed and across how many replicas, placing the signal between code coverage, which distinguishes too few schedules, and complete trace coverage, which distinguishes too many. It requires no executable specification, model checker, or correspondence between implementation events and model actions. Instead, it relies on information available from the execution trace, principally which transaction and protocol phase each message belongs to.

3.2.2. Motivating Example

The linearizability violation described in Section 2.5 illustrates the information that such predicates should preserve. Reproducing the bug requires a Slow Path commit, a coordinator crash after local execution but before commit propagation, a recovery quorum containing a replica unaware of the executed transaction, and a second recovery quorum excluding every replica that learned the outcome of the first recovery. Each condition concerns a phase a transaction has reached, a number of replicas that have or have not reached it, or a relation between sets of replicas. None requires the concrete value of a timestamp, ballot, or dependency set, even though the protocol maintains all three and the violation ultimately concerns transaction ordering.

This is not merely a feature of the informal description. Accord's recovery decisions are expressed using the same kinds of information. Fast Path recovery in Section 2.4 depends on whether a sub-quorum of $|Q| - e$ replicas preaccepted a transaction with its initial timestamp unchanged, while the choice between recovery branches depends on the most advanced phase reported by the quorum. These observations suggest that useful abstract states can be formed from how far each transaction has progressed and across how many replicas.

3.2.3. Design

Guided by this motivating example, we construct the predicate abstraction from the progress of requests and their replies. Consider a single transaction in a cluster of seven replicas at the point where its coordinator broadcasts PreAccept. A quorum here is four replicas. The counts that follow are taken over network messages, and a coordinator does not send a network message to itself (for simplicity), so a round that reaches every other replica can register six rather than seven. The abstraction records two numbers about this round, and the remainder of this section builds the coverage measure out of them.

The first is the number of distinct replicas to which the PreAccept has been delivered, and the second is the number of distinct replicas whose reply has been delivered back. Writing the pair as $(sent, acked)$, a schedule that delivers three requests, then two replies, then a fourth request, then two further replies, then the remaining two requests and their replies takes the transaction through $(0, 0)$, $(2, 0)$, $(3, 0)$, $(3, 2)$, $(4, 2)$, $(4, 4)$, $(6, 4)$, and $(6, 6)$. The second number is not determined by the first, since a schedule is free to deliver all six requests and none of the replies, leaving the round at $(6, 0)$, and it is precisely this kind of difference that reordering the schedule produces.

Algorithm 1 summarises how these counts are extracted from a trace. The map `counts` associates each transaction and stage pair with two sets, `sent` and `acked`. `EXTRACTSTAGE` obtains the transaction and stage from the delivered message. For generic replies that do not carry this information directly, it uses the request identifier to consult the corresponding request. Sets ensure that repeated or duplicated deliveries involving the same replica do not increase a count.

Algorithm 1 Updating stage counts from a delivered message

```

1: function UPDATESTAGECOUNTS(event, counts)
2:   (transaction, stage) ← EXTRACTSTAGE(event)
3:   (sent, acked) ← counts[transaction, stage]
4:   if event is a request then
5:     sent ← sent ∪ {event.receiver}
6:   else if event is a reply then
7:     acked ← acked ∪ {event.sender}
8:   end if
9:   counts[transaction, stage] ← (sent, acked)
10:  return counts
11: end function

```

Recording the counts themselves would treat every delivery as equally significant, which does not match what the protocol does with them. The step from $(3, 3)$ to $(4, 4)$ crosses the quorum and changes what the coordinator is able to do next, whereas the step from $(2, 0)$ to $(3, 0)$ corresponds to no decision the protocol makes. Counting exactly also makes the number of distinguishable rounds grow with the size of

the cluster without distinguishing anything further. Each of the two numbers is therefore classified into one of four values, namely none when it is zero, subset when it is positive but below the quorum, quorum when it is at or above the quorum but short of every addressable replica, and all when it has reached every addressable replica. A stage is then described by the pair of classes rather than by the pair of counts. Under this classification the sequence above passes through (NONE, NONE), then (SUBSET, NONE) where it stays for two deliveries, then (SUBSET, SUBSET), (QUORUM, SUBSET), (QUORUM, QUORUM), (ALL, QUORUM) and finally (ALL, ALL).

A coarser description of the same round is available by reducing each pair to two booleans, one recording whether the stage has been reached at all and one recording whether it has completed, in the sense that both counts have arrived at a quorum. The eight steps above then collapse to three distinct values, one before the first delivery, one covering everything up to the point where both counts reach four, and one from there onward. This is not a second abstraction so much as a second reading of the same two counters, and the same holds of the counts themselves, which can be used unclassified. The three levels differ only in how much of the counters they retain, so the resolution of the abstraction is a parameter rather than a commitment, and the remaining design decisions are unaffected by it.

Algorithm 2 gives the two classification rules evaluated in this thesis. Here, n denotes the number of other replicas that can be observed through network messages, rather than the total cluster size.

Algorithm 2 Classifying the request and acknowledgement counts of one stage

```

1: function CARDINALITYCLASS(count, quorum, n)
2:   if count = 0 then
3:     return NONE
4:   else if count < quorum then
5:     return SUBSET
6:   else if count < n then
7:     return QUORUM
8:   else
9:     return ALL
10:  end if
11: end function

12: function CLASSIFYSTAGE(sent, acked, mode, quorum, n)
13:  if mode = Extent then
14:    sentClass ← CARDINALITYCLASS(sent, quorum, n)
15:    ackedClass ← CARDINALITYCLASS(acked, quorum, n)
16:    return (sentClass, ackedClass)
17:  else if mode = ReachedCompleted then
18:    reached ← (sent > 0)
19:    completed ← (sent ≥ quorum ∧ acked ≥ quorum)
20:    return (reached, completed)
21:  end if
22: end function

```

Attributing a delivery to a transaction and a stage requires nothing beyond the name of the message type. Accord's messages follow a convention under which a request is named for the phase it belongs to and its reply is named after the request it answers, so the stage a delivery contributes to can be read off the type name directly, without a table written by hand and without any knowledge of what the receiving replica does with the message. The transaction is read from the message in the same way. A small number of replies are generic and do not name a stage, and these are correlated back to the request they answer, which Chapter 4 describes. This is what makes the abstraction cheap to apply, and it is also what makes it portable, since nothing in the construction refers to Accord beyond the names of its messages, and any protocol that proceeds in phases and names its messages accordingly can be instrumented the same way. Accord is the system this work evaluates the abstraction on rather than the system the abstraction is built around.

3.2.4. Predicate State Machine

The description so far covers one stage of one transaction. A transaction touches several stages over its lifetime, and each of these carries its own pair of counts, classified independently. An execution involves many transactions, since the harness keeps a fixed number in flight at any time, and each of these carries its own set of stages. The abstract state of an execution at a given point is therefore the joint valuation over every transaction observed so far, each described by the classification of every stage it has touched. Alongside this, the valuation records which replicas are currently crashed, taken as the set of replicas that have crashed without having since restarted. Crashes are included for the same reason as the stage counts: whether a crash occurs before, during, or after a transaction's progress through a stage produces a different situation, and this is a difference the schedule controls in exactly the way it controls the order of deliveries.

This joint valuation is not evaluated once at the end of an execution but after every event in the trace. The coverage of an execution is the set of distinct valuations it passed through, and an execution that arrives at a given valuation by a different route than an earlier one contributes the intermediate valuations along that route. Consider two transactions each reaching a quorum on PreAccept. An execution in which the first completes the stage before the second begins passes through a valuation in which one transaction is at quorum and the other has not been reached, while an execution in which the two advance together passes through a valuation in which both are at subset. The two agree once both have completed, so a measure that only inspected the final valuation would not separate them. The order of deliveries is what the fuzzer's mutations perturb, and sampling after every event is what makes that order visible to the coverage measure.

Algorithm 3 summarises this incremental replay. The helper `RELEVANTSTAGES` returns every stage reached by a transaction, while `CURRENTSTAGES` returns its highest backbone stage and a separate recovery stage. `CLASSIFYTRANSACTION` applies Algorithm 2 to the selected stages. The variable `transactionStates` contains the resulting summary of every transaction observed at the current trace prefix. Combining these summaries with the set of crashed replicas produces `currentState`, the joint coverage state after the current event. The variable `previousState` stores the joint state produced after the preceding event, allowing events that do not change the abstraction to be skipped. Finally, `visitedStates` is the set of all distinct joint states reached during the execution and therefore represents the predicate coverage of the schedule. `BUILDSTATEKEY` sorts transactions, stages, and crashed replicas so that two equivalent abstract states receive the same coverage key regardless of insertion order.

Algorithm 3 Collecting the joint predicate states visited by an execution

```

1: function COLLECTPREDICATECOVERAGE(trace, classifier, historyMode)
2:   visitedStates  $\leftarrow \emptyset$ 
3:   crashed  $\leftarrow \emptyset$ 
4:   counts  $\leftarrow$  empty map
5:   previousState  $\leftarrow$  UNDEFINED
6:   for all event  $\in$  trace do
7:     if event is a message delivery then
8:       counts  $\leftarrow$  UPDATESTAGECOUNTS(event, counts)
9:     else if event = Crash(node) then
10:      crashed  $\leftarrow$  crashed  $\cup$  {node}
11:     else if event = Restart(node) then
12:      crashed  $\leftarrow$  crashed  $\setminus$  {node}
13:     end if
14:     transactionStates  $\leftarrow []$ 
15:     for all transaction observed so far do
16:       if historyMode = FullHistory then
17:         stages  $\leftarrow$  RELEVANTSTAGES(transaction)
18:       else
19:         stages  $\leftarrow$  CURRENTSTAGES(transaction)
20:       end if
21:       transactionState  $\leftarrow$  CLASSIFYTRANSACTION(transaction, stages, classifier, counts)
22:       append transactionState to transactionStates
23:     end for
24:     currentState  $\leftarrow$  BUILDSTATEKEY(crashed, transactionStates)
25:     if currentState  $\neq$  previousState then
26:       visitedStates  $\leftarrow$  visitedStates  $\cup$  {currentState}
27:       previousState  $\leftarrow$  currentState
28:     end if
29:   end for
30:   return visitedStates
31: end function

```

Under the definition above, a transaction accumulates entries as it progresses, so one that has reached the commit stage still carries the classification of its PreAccept stage, and a reply to that earlier round arriving late will change the valuation even though the coordinator has long since moved past the point where it could act on it. The alternative is to retain only the stage a transaction currently occupies, dropping the entries for stages it has left behind. Since a stage once touched is never untouched, the current stage under this reading is a non-decreasing watermark over the protocol's stage ordering, and a late or duplicated delivery for a superseded stage cannot move it backwards.

Recovery does not fit into that ordering. A transaction may enter recovery at any point in its life, and having done so is not progress beyond the stage it had reached, so there is no position in the sequence at which a recovery entry could sit without either displacing a stage that is still current or being displaced by one. Under the accumulating reading, this raises no difficulty, as recovery is one more stage among the rest and is retained alongside them. Under the watermark reading, a transaction is therefore described by two entries rather than one: the classification of its current stage and, separately, the classification of its recovery activity, with the latter recorded as absent for transactions that never entered recovery. The classification level and the amount of retained history form two independent parameters of the abstraction.

3.2.5. Evaluated Configurations

The evaluation considers three configurations that vary the two independent parameters introduced above: the resolution with which stage progress is classified and the amount of transaction history retained. Comparing Extent with Reached/Completed isolates the effect of classification resolution,

since both retain full history. Comparing Extent with Collapsed History isolates the effect of retaining earlier stages, since both use the same four-level classification.

Extent

Extent is the most detailed configuration evaluated. For every stage reached by a transaction, it classifies both requests and acknowledgements as `NONE`, `SUBSET`, `QUORUM`, or `ALL`, and retains these classifications after the transaction advances to later stages. Its purpose is to preserve the protocol boundaries at which a round begins, reaches only part of the cluster, crosses a quorum, or reaches every addressable replica, without distinguishing every exact count. In the motivating execution of Section 3.2.2, Extent records that τ reaches only a `SUBSET` of replicas during `PreAccept`, while γ progresses through the Slow Path and its decision is only partially propagated before the coordinator crashes. Because earlier stages remain present, the resulting state also preserves that γ passed through `Accept` before later entering recovery and committing again.

Reached/Completed

Reached/Completed tests whether a much coarser description of progress is sufficient. It retains the full stage history used by Extent, but represents each stage using only two booleans: whether any request for the stage has been delivered and whether both its request and acknowledgement counts have reached a quorum. It therefore preserves which phases occurred and which completed, while discarding the distinction between different `SUBSET` extents and between `QUORUM` and `ALL`. In the motivating execution, τ 's `PreAccept` stage is recorded as reached but incomplete. However, every positive delivery count below the quorum receives the same representation, so the configuration cannot distinguish whether τ was observed by one replica or by a larger `SUBSET`. This configuration tests whether those finer distinctions provide useful guidance or merely enlarge the coverage space.

Collapsed History

Collapsed History tests the second design parameter: whether progress through earlier stages remains useful once a transaction advances. It uses the same `NONE`, `SUBSET`, `QUORUM`, and `ALL` classes as Extent, but retains only the transaction's highest backbone stage rather than every stage it has reached. This reduces the size of each state and prevents late messages from superseded stages from changing coverage indefinitely. In the motivating execution, once γ advances from `PreAccept` to `Accept`, its `Accept` summary replaces its `PreAccept` summary rather than being retained alongside it. Transaction τ remains represented by `PreAccept` because it has not advanced beyond that stage. Comparing this configuration with Extent tests whether the discarded stage history is important for directing exploration.

Chapter 5 compares the coverage produced by these three configurations.

4

Implementation

The method was implemented entirely within `accord-core` by extending Accord's existing `BurnTest` framework¹ with an additional suite of Java classes. The implementation is available in the accompanying repository.² The fuzzer itself runs as a JUnit test, using `BurnTest` to execute the production protocol in a simulated environment. This chapter first introduces `BurnTest`, then describes the scheduler, the mapping from executions to abstract states, and the guidance loop.

4.1. `BurnTest`

Accord ships with an integrated testing framework, `BurnTest`, which exercises protocol executions under randomised but deterministic conditions. The test runs as a loop within a fully simulated environment. The message handling, consensus logic, and recovery protocol use the production implementation, while the network, passage of time, and scheduling are replaced by simulated equivalents.

Determinism is one of the key properties that makes `BurnTest` a viable foundation for fuzzing. It is achieved by sampling all random variables from a single source with a predefined seed. Therefore, all message delays, transaction keys and IDs, and network jitter can be reliably reproduced across runs. The network itself is simulated by a custom queue that reorders messages by assigning each one a random delay, creating reorderings and partitions without explicit fault injection. Concurrency is bounded by an input parameter. The harness keeps a fixed number of transactions in flight, immediately submitting a new one whenever a previous transaction completes.

Another advantage of `BurnTest` is that everything the test processes sits on one "global" queue, ordered by scheduled time. Network messages share this queue with internal runnables and scheduled callbacks such as command store tasks, progress polling, and other timers that drive recovery. There is no separate network layer, time advancement, or thread scheduler. A timeout firing and a message arriving are both queue entries with a future timestamp, dispatched by a single loop. In practice, this structure ensures that the harness can execute large numbers of schedules quickly, since no real-time delays are ever incurred.

However, this shared structure also restricts how the queue can be controlled. Internal runnables must keep progressing or the simulation stalls. If scheduled tasks are starved, timeouts never fire, transactions block waiting for replies, and the system halts without ever producing a useful execution. The natural design choice is therefore minimal intervention, reordering only network messages relative to each other while letting the rest of the queue advance under its own ordering. This is the constraint within which the scheduling detailed in the next section operates, and this control introduces its own decisions and trade-offs.

¹<https://github.com/apache/cassandra-accord/tree/trunk/accord-core/src/test/java/accord/burn>

²<https://github.com/St-Stefan/cassandra-accord-fuzz>

4.2. Scheduling

The main difficulty in exercising control is one of abstraction. ModelFuzz [10] reasons about an execution in terms of three event types, namely delivering a message from a given link's buffer, crashing a node, and restarting it. BurnTest, on the other hand, exposes a single queue of heterogeneous entries. Much of this section describes an approach that bridges these two views so that a schedule expressed using ModelFuzz's actions can be replayed with BurnTest's queue.

Recording a schedule is the simpler task. Here, the scheduler is a passive observer, delegating every poll to the underlying queue and appending each delivered network message to a trace. In our implementation, a trace is simply an ordered list of delivery events, where each event stores the sender, receiver, message type, and other protocol fields needed to identify it later. Runnables are not recorded, since they are internal to a node and are not part of the schedule we aim to control. Every run produces a trace in this way, whether or not it follows a previously recorded one, which means the result of a guided execution can itself be mutated and replayed.

At every poll, the scheduler must choose which entry to return from the queue. When recording, this choice is trivial, as the scheduler simply returns whatever the queue provides. During replay, however, it is constrained by the trace, and this is where the difficulty lies. The next event in the trace names a specific network message that should be delivered, but the queue holds a mix of network messages and runnables, ordered only by their scheduled time. The expected message is rarely at the front, so the central question becomes what the scheduler should do when the entry it wants is not the entry it is given.

The first decision is the order in which the two kinds of entry are considered. One natural option is to poll runnables first, on the grounds that they keep the simulation alive. This does not work in practice. Internal runnables include the timers that drive recovery, and these tick faster than the queue ever empties, so polling them first means that almost every poll returns a runnable and the scheduler never gets a chance to act on a network message. The order is therefore reversed. The scheduler first attempts to find the expected packet, returns it if it is present, and only falls through to running a runnable when no usable packet is available. Finding that packet, however, means scanning the queue and pulling every network message from it in the process. This is an opportunistic way to operate, sitting somewhat at odds with the minimal-intervention goal described above.

In practice, this turns out to be less invasive than it appears. Anything sitting on the queue has already been emitted by the protocol, so draining the packets into the buffer neither invents deliveries nor discards them. Additionally, delivering a chosen packet causes the receiving node to schedule new runnables, and because the queue is ordered by time, these generated entries sit ahead of the pending timeouts.

The buffer is also what allows BurnTest's single queue to be read as the per-node buffers that ModelFuzz expects. A ModelFuzz delivery does not name a specific message [10]. A delivery action means delivering the oldest message currently waiting for node 5, taking the front of that node's buffer in FIFO order. BurnTest's queue does not expose this directly, since it interleaves messages for every node in scheduled-time order. To recover the per-link view, the scheduler pulls packets from the queue during its scan and appends them to the buffer in the order they were encountered. The buffer therefore acts as a set of ordered per-link queues, and delivering the message for a given sender and receiver pair amounts to taking the oldest packet on that link, exactly as ModelFuzz would. A packet pulled from the queue waits in the buffer until it is the oldest match for the link on which the trace requests a delivery.

One case remains, namely when the trace asks to deliver on a link that has nothing waiting for it. This happens naturally during replay, since a mutation could call for a delivery for which the protocol has not yet produced a message, or for which it never produces one at all. The link may be empty only for a moment, with a message still to arrive once some runnable fires, or it may stay empty indefinitely. The scheduler cannot tell these cases apart, so it waits. A bounded window of logical time, five hundred milliseconds in the current implementation, is allowed to pass while the rest of the queue advances. If a message appears on the link within that window, it is delivered as normal; if none does, the delivery is treated as an empty-buffer action and replay moves on to the next event. This mirrors the way ModelFuzz works on a real system, delivering whatever is present and simply returning nothing when a buffer is empty. Without this window, a single impossible delivery would block replay forever.

4.3. Mapping Executions to Abstract States

The guidance strategy described in Section 3.1 depends on turning a recorded execution into a sequence of abstract states, and therefore needs a faithful translation layer between BurnTest messages and TLA+ actions. The various design choices taken to implement such a layer are described in the present section. This is also where the general cost identified in Chapter 3 takes a concrete form, since Accord’s specification and its implementation were written at different points in time and do not correspond one-to-one.

We first describe the ways in which the two do not line up, and then the mechanisms used to bridge them. These mechanisms are not equally clean. Some follow directly from how the implementation encodes the protocol and preserve the meaning of the execution exactly. Others recover information that an execution contains but that no single message carries. Others still impose a choice that the implementation does not make, and are faithful only up to that choice. Presenting them in order of the assumptions they require makes clear how much of the resulting coverage measure rests on solid ground and how much rests on modelling decisions.

Table 4.1 lists the concrete Accord messages and events handled by the mapper and the TLA+ actions they produce. The rows are grouped by the mapping case discussed in the following subsections.

Table 4.1: Mapping Accord messages and events to TLA+ actions.

Message or event	Mapping
Action–Message Discrepancy	
PreAccept	HandlePreAcceptWrapper
PreAcceptOk	HandlePreAcceptOK
Accept	HandleAcceptWrapper
RecoverOk	HandleRecoverOK
Apply	Execute
ReadTxnData, ReadOk, ReadOkWithFutureEpoch	Omitted
One Message, Several Actions	
Commit	HandleCommitWrapper; HandleStableWrapper
StableThenRead	HandleCommitWrapper; HandleStableWrapper
Actions Without a Message	
BeginRecovery	StartRecover; HandleRecoverWrapper
Client submission	Submit
Recovering Absent Correlations	
AcceptReply	HandleAcceptOK

4.3.1. Action–Message Discrepancy

The corresponding mappings are listed under *Action–Message Discrepancy* in Table 4.1. The most basic obstacle is that the specification and the implementation advance in different units. The TLA+ model is a state machine whose transitions are named actions parameterized over the constant sets of the specification, so that a step is an instance such as `HandlePreAccept( $p, id$ )` or `Submit( $p, id$ )` for a specific process and transaction. TLC advances by firing one such enabled action at a time. The implementation, by contrast, produces no actions at all. It produces a stream of concrete message deliveries, each an ordinary network message arriving at a node. A recorded schedule is therefore a list of deliveries, and to obtain a run of the model each delivery must be turned into the action instance it corresponds to.

For most messages this correspondence is direct. A delivered PreAccept becomes HandlePreAccept at the receiving process for the transaction the message concerns, an Accept becomes HandleAccept, and so on for the remaining protocol messages. The action’s parameters are read straight off the delivery, since the receiving process and the transaction identity are both present on the message. In this common case the mapping is a lookup, matching the message type to the corresponding action and filling in the parameters from the message fields.

Two further facts complicate this base case, and each is the subject of a subsection below. First, the correspondence is not always one message to one action. A single delivery can expand into several actions, and a single Java message class can correspond to different specification actions depending on its contents, so the mapping cannot be a plain type-to-type table. Second, some specification actions

correspond to no delivered message at all and must be synthesised by the mapping rather than read from the trace. A separate and simpler point concerns messages the specification does not model. These cannot be turned into actions, and rather than emitting an empty action, the mapping omits them. Some, such as durability traffic, never enter the recorded schedule, while others are recorded but dropped when the schedule is translated, so TLC receives only steps that the model can take.

4.3.2. One Message, Several Actions

The corresponding mappings are listed under *One Message, Several Actions* in Table 4.1. The specification treats committing a transaction and marking it stable as two distinct actions, `HandleCommit` followed by `HandleStable`. The implementation preserves this on the slow path, where a `Commit` message and a later `Stable` message arrive in sequence, but on the fast and medium paths it folds both into a single `Stable` message that also carries the commit decision. Such a delivery is expanded into the two actions the specification expects, `HandleCommit` and then `HandleStable` for the same process and transaction. A field on the message records the path taken, so that a `Stable` on a fast or medium path is expanded while a `Stable` on the slow path, already preceded by its own `Commit`, is mapped alone.

The correspondence also varies at the level of message classes. The implementation uses one `Commit` class to carry both the commit decision and, on the combined paths, the stable decision, distinguishing them only by an internal kind field. The same class therefore maps to `HandleCommit` in one case and `HandleStable` in another, and the mapping reads that field to decide which. The message type alone does not identify the action.

The last case maps a delivery to an action of a different name. Once a transaction is stable the implementation delivers an `Apply` message to trigger its execution, whereas the specification models execution as a dedicated `Execute` action rather than the handling of a message. An `Apply` delivery is therefore mapped to `Execute` at the receiving process.

Each of these involves deeper instrumentation. The path marker, the kind field, and the message type are all present in the delivered message and therefore nothing is reconstructed. The mapping only has to know how the implementation encodes what the specification keeps separate.

4.3.3. Actions Without a Message

The corresponding mappings are listed under *Actions Without a Message* in Table 4.1. The opposite problem is an action that the specification takes but no message triggers. Recovery begins in the specification with `StartRecover( $p, id$ )`, an action a process performs of its own accord when it suspects a coordinator has failed. Nothing is received to prompt it. In the implementation, the same event appears only indirectly, as the recovery coordinator begins to send its `Recover` messages, and so the trace records the resulting `Recover` deliveries but never the decision to start recovering.

To run the model, that decision has to be restored. When a `Recover` delivery is seen from a coordinator that has not yet been observed recovering the transaction in question, a `StartRecover` action is synthesised for that coordinator and transaction before the `Recover` delivery itself is mapped. A record of which coordinators have already started recovering each transaction ensures that the action is synthesised once per recovery rather than once per `Recover` message. Unlike the cases above, this step adds an action that no single delivery carries, inferring it from the presence of the messages that recovery produces.

4.3.4. Controlled-Mode Specification Changes

For TLC to drive a run under external control, it must enumerate the enabled actions at each state and offer them to the fuzzer by name. It can only do so for actions quantified over constant sets. In the original specification the message handlers are quantified over the message pool, a state variable, so TLC cannot name them and collapses them into a single anonymous action invisible to the mapping. The specification is therefore restructured so that each message handler is wrapped in an operator parameterized over the constant sets of processes and transactions, with the quantification over messages moved inside the operator body. TLC can then enumerate and name every handler.

The restructuring changes no protocol logic, no invariant, and no reachable state. Each wrapper is enabled by exactly the messages that enabled the original handler, since the conditions the wrapper adds are already implied by the handler's own preconditions.

4.3.5. Recovering Absent Correlations

The corresponding mapping is listed under *Recovering Absent Correlations* in Table 4.1. A specification action is identified by its process and transaction, both of which must be read from the delivered message. For most messages they are present. The exception is the reply to an Accept, which does not carry the identity of the transaction it concerns, so the transaction cannot be read from the reply alone.

The missing identity is recovered by correlation. Each request is tagged with a stable identifier when it is sent, and each reply carries the identifier of the request it answers, so a table from identifier to transaction lets the reply be resolved to the transaction associated with that request. The identifier is derived from fields already stable across runs, so the same request carries the same identifier regardless of the order of surrounding events, which allows replies to be matched consistently even after a schedule has been mutated.

The assumption here is mild. The correlation being reconstructed genuinely holds in the execution, and the identifier is a faithful key for it. Nothing is chosen or approximated. The mapping only rebuilds an association that the reply, taken alone, does not expose.

4.3.6. Identities and Timestamps

The specification identifies transactions by small integers and orders them by a timestamp rank, neither of which is directly present in the execution. Transactions in the implementation carry hybrid-logical-clock identifiers that are neither small nor contiguous, and therefore the mapping must assign each one a stable integer identity and rank before a run can be replayed against the model.

Both are assigned by imposing an order. Transactions are ordered by their identifier to fix a stable integer identity, and by their timestamp to fix the rank the specification uses to break ties. The assignment is deterministic, so the same execution always yields the same identities, which is what lets a recorded schedule be replayed and mutated consistently.

4.4. Guidance

Exploration works by looping over a work queue of schedules, initialized with a population of freshly generated executions produced by running the harness while recording the resulting delivery order. These seeds are recorded executions rather than synthetic schedules and are as long as an unconstrained run of the harness happens to be. Every iteration draws one schedule from the queue and replays it through the guided queue of Section 4.2. The trace recorded during that replay is then scored, and mutants are derived accordingly.

Scoring is performed by a coverage signal. A signal maps every state reached by an execution to a hash and reports how many of those hashes have not been seen in any earlier iteration of the run. Two signals are implemented. The abstract-state signal of Section 3.1 replays the mapped action sequence against the specification and hashes the model states it visits. The predicate signal of Section 3.2 derives its states directly from the recorded trace. Both report the number of previously unseen states in the current execution and the cumulative number seen so far, and both are evaluated on every iteration regardless of which one is in control. Only one supplies the number that drives mutation, and that choice is the only difference between the two guided strategies.

Let $v(s)$ denote the number of previously unseen keys reported for an execution s by whichever signal is in control, let m be the number of mutants generated per unit of novelty, and let Q be the capacity of the work queue and $|W|$ its current occupancy. The number of mutants produced from s is

$$e(s) = \max\left(0, \min(m \cdot v(s), Q - |W|)\right).$$

An execution reporting no new keys therefore produces no mutants and leaves no descendants, which is the novelty gate that makes the search selective, while an execution that does report new keys receives a proportional number of mutants, capped by the room left in the queue. Note that $v(s)$ is an absolute count and is not normalised by the length of the execution. Two schedules that produce the same number of new states receive the same allowance regardless of how many events each took to produce them.

Every mutant is derived from a schedule already in the queue, so the queue fills up over time with descendants of the initial population. An execution that is not one of those descendants enters the search in only two ways. When the queue is empty, the iteration runs a fresh unconstrained execution in place of a replay. Separately, a periodic reseed can be configured to add a new population at a fixed iteration interval, and this is disabled in the long-running configurations. The rate at which fresh executions enter is therefore set by the guidance signal itself, since the queue empties only once executions stop reporting new states.

Experimental Evaluation

This chapter evaluates TLA+ model-guided fuzzing, predicate-guided fuzzing, and two unguided baselines called Unguided Mutation and Random. All experiments exercise the single-partition protocol described in Chapter 2 through the controlled BurnTest queue of Section 4.2. The evaluation follows the research questions defined in Section 1.1. Section 5.2 addresses RQ1, Section 5.3 addresses RQ2, and Section 5.4 addresses RQ3. Section 5.6 then discusses anomalous behaviour and the limitations of the experimental design.

5.1. Experimental Setup

5.1.1. Protocol Configurations and Workload

Experiments use clusters of $n = 5$, $n = 7$, and $n = 9$ replicas. The five- and seven-replica configurations inject one crash-stop fault, while the nine-replica configuration injects two. At $n = 5$ the fast-path quorum coincides with a simple majority, and the protocol reaches the Slow Path comparatively rarely. At $n = 7$ and $n = 9$ the fast-path quorum is larger than a bare majority, causing the Accept exchange and the recovery of Slow Path state to be exercised more frequently. The workload is fixed across strategies and transactions are drawn from the same distribution of keys and conflicts.

Results are reported across the three cluster sizes where available, but the detailed analysis focuses on the seven-replica configuration.

5.1.2. Execution Environments

Most experiments were conducted as single-machine experiments on CloudLab. One TLC server served at most four concurrently running fuzzer instances on the same machine. Predicate-guided experiments and long-duration ModelFuzz experiments were run on a MacBook Air with an M3 processor. The implications of shared resources and the use of two execution environments are discussed in Section 5.6.5.

5.1.3. Fuzzer Configuration

Each guided run begins with 40 randomly generated seed schedules. The work queue holds at most 700 schedules. Schedules are limited to 500 events in the five and seven-replica configurations and to 1000 events in the nine-replica configuration. Unless stated otherwise, experiments stop at a fixed wall-clock cutoff rather than after a fixed number of iterations. Results are measured against fuzzer iterations; wall-clock time is used only as the stopping condition.

The Unguided Mutation baseline starts from the same seed schedules and applies one randomly selected mutation per iteration to either a seed or a schedule produced by an earlier mutation. The resulting schedules form chains of descendants without being selected according to coverage. The Random baseline instead generates a fresh independent schedule on every iteration.

The averaged plots in this chapter use five independent runs per method, initialized with different random seeds, with each run capped at 24 hours of wall-clock time. Curves may therefore terminate at

different iteration counts because the methods have different per-iteration costs.

5.1.4. Evaluation Metrics

Coverage is measured as the number of unique abstract states visited in Accord’s TLA+ model, with message contents and timestamps abstracted away such that executions that differ only in these fields are counted as equivalent. The same metric is applied to all four methods, including predicate-guided exploration and the two unguided baselines. The model-state mapping has already been implemented for the ModelFuzz baseline, making it available as a common external measure even when it is not the signal guiding the run.

For fault-finding experiments, effectiveness is measured by whether a run reaches the violating execution identified in Section 5.2. Where repeated runs are available, the comparison reports the number or proportion of runs that reach the violation and the amount of testing required before it is first observed.

5.2. Fault Discovery and Characterization

This section addresses RQ1. It describes the correctness violation exposed during controlled exploration, the property that is violated, and the sequence of message deliveries, crashes, and recovery decisions required to produce it. The analysis also distinguishes behaviour originating in Accord from artefacts introduced by the scheduler or execution harness. The failure reported here is distinct from the published violation described in Section 2.5; it was discovered independently rather than by replaying that execution.

During an Unguided Mutation run, a seven-replica execution terminated with an assertion wrapping an `IllegalStateException` raised by `Command.validate`. Replayng the surviving schedule reproduces the failure on the current Accord implementation. The failed check requires a command’s `executeAt` timestamp to be strictly greater than its transaction identifier, except for a restricted equality case. The exception is raised while applying a Slow Path Accept message, showing that the implementation reaches a command state that violates its own timestamp-ordering invariant. This is a reproducible internal correctness failure, although the current evidence does not show a client-visible linearizability violation.

The minimized replay input contains only eight controlled events. It submits a transaction to coordinator n_2 , marks n_2 as crashed during part of the PreAccept fanout, and clears the crash marker before the remaining PreAccept messages are processed. This removes five recovery markers from the earlier reproducer that targeted nodes which had never been crashed and were therefore no-ops. The full execution generated from the shorter prefix contains 214 recorded events, numbered 0 through 213.

The resulting execution leaves the first transaction unevenly observed across the replicas. Accord’s progress mechanism later causes n_4 to recover this transaction. At the same time, n_2 and n_5 independently initiate competing recoveries for a later transaction. The recovery led by n_4 obtains the necessary responses and broadcasts Accept on the Slow Path for the first transaction, immediately before the invariant failure is surfaced.

The trace supports an association between partial PreAccept delivery, overlapping recovery activity, and the invalid `executeAt` state. It does not yet identify the replica on which the exception is raised or the exact sequence of local command-state updates that violates the invariant. We therefore treat the recovery activity as the observed execution context rather than as a confirmed root cause. Appendix A gives the controlled prefix and a condensed account of the full trace.

5.3. Fault-Finding Effectiveness

This section addresses RQ2 by comparing how often the four methods reach the invariant failure identified above. The comparison treats observing the exception as the common outcome, regardless of the coverage signal used to guide each run.

Figure 5.1 reports the first iteration at which the exception was observed during five 24-hour runs per method. Unguided Mutation reached the exception in all five runs. ModelFuzz and predicate guidance each reached it in one of five runs, while Random did not reach it in any run. In the successful guided runs, predicate guidance reached the exception after only a few thousand iterations and ModelFuzz

after approximately 20,000 iterations. The successful Unguided Mutation runs varied substantially, from fewer than 10,000 to more than 100,000 iterations.

The bars and error bars in the figure are calculated only over runs that reached the exception. Consequently, the low iteration count of the single successful predicate run must not be interpreted as evidence that predicate guidance is generally the fastest method. The more informative observation in this experiment is the hit rate, for which Unguided Mutation performs best. The small number of seeds and its unexpected advantage are discussed in Section 5.6.4.

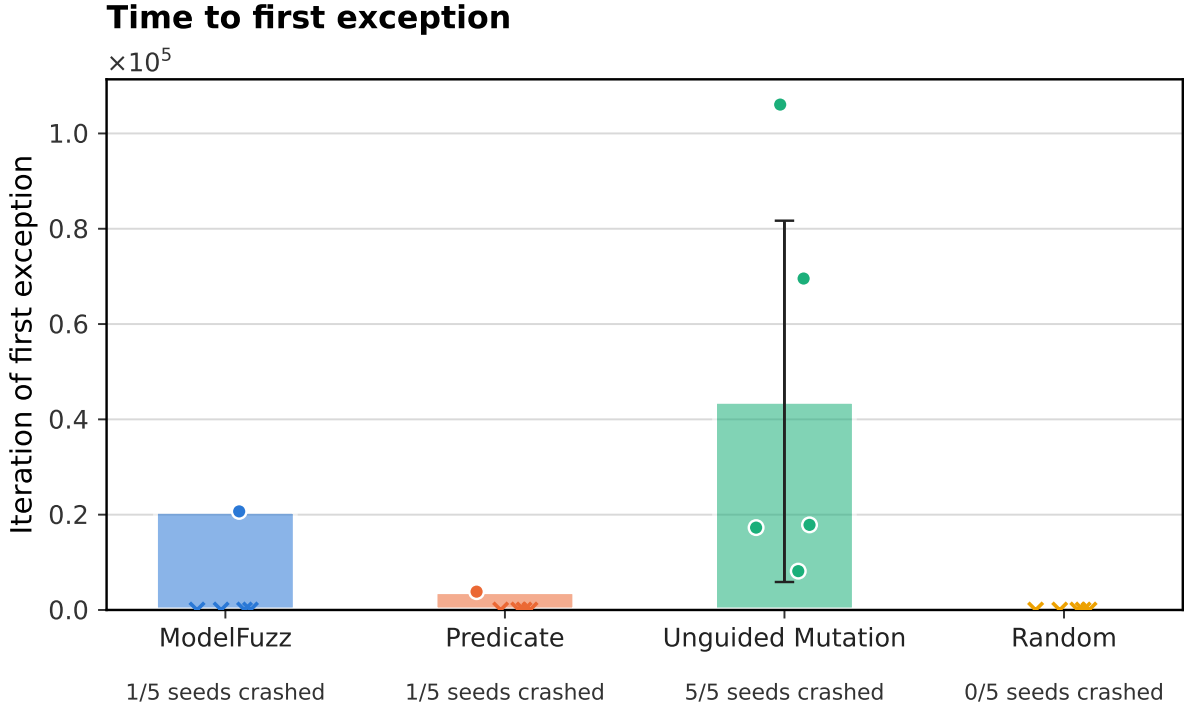


Figure 5.1: Iteration of the first observed exception across five 24-hour runs per method. Bars show the mean and one standard deviation over successful runs only; the labels below the axis report how many runs reached the exception. Crosses denote runs in which no exception was found.

5.4. Abstract-State Coverage

This section addresses RQ3. Figure 5.2 reports the mean number of unique abstract TLA+ states discovered over five 24-hour runs of the seven-replica configuration. Since the horizontal axis measures fuzzer iterations, the figure compares coverage efficiency per executed schedule rather than coverage per unit of wall-clock time.

ModelFuzz discovers substantially more abstract states than the other methods, reaching approximately 2.9 million states near the end of its runs. Predicate guidance and Unguided Mutation initially remain closer together. Predicate guidance overtakes Unguided Mutation at approximately 100,000 iterations and ends at roughly 1.15 million states, compared with approximately 1.0 million for Unguided Mutation. Random discovers the fewest states, ending near 550,000 despite completing the largest number of iterations. The dashed portions of the curves indicate that fewer than five seeds remained active at those iteration counts because the runs were capped by wall-clock time.

5.4.1. Predicate Abstraction Variants

We separately compare the three predicate configurations defined in Section 3.2.5. They are Extent, Reached/Completed, and Collapsed History.

Figure 5.3 measures each configuration using the predicate states that guide that configuration. Extent continues discovering new states throughout the experiment and reaches approximately 1.25 million states. Collapsed History grows to roughly 120,000 states before largely flattening, while

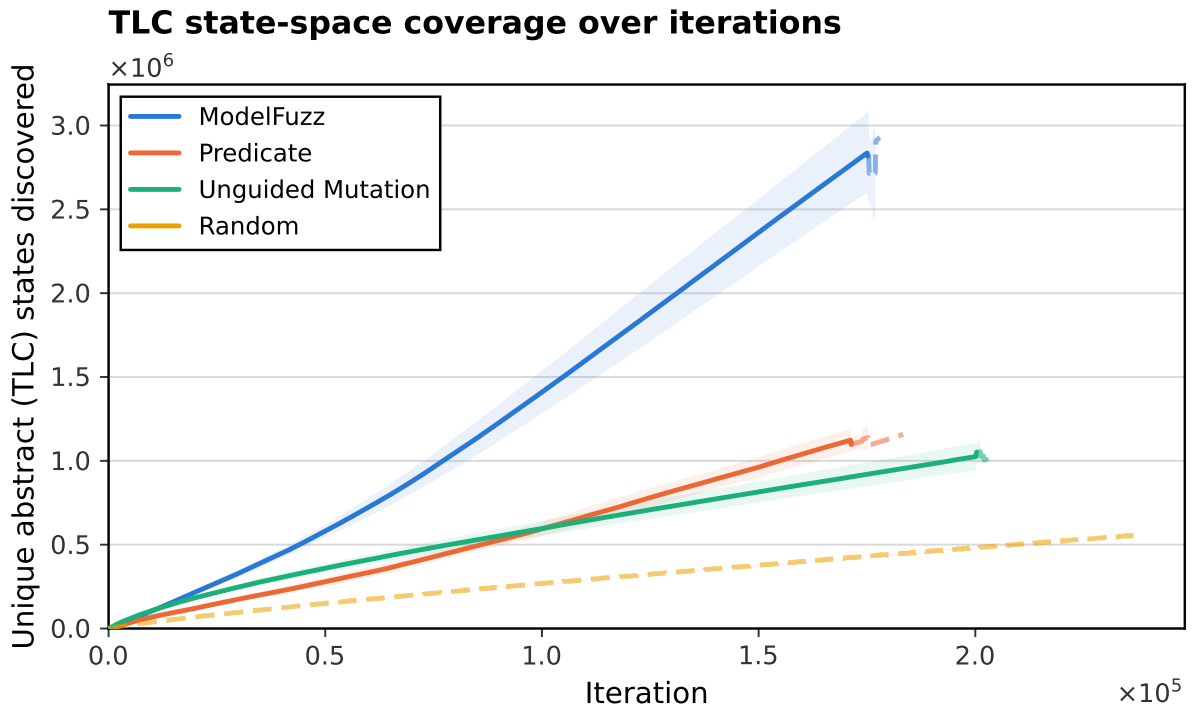


Figure 5.2: Mean unique abstract TLA+ states discovered over fuzzer iterations, with one standard deviation across five seeds per method. Each run is capped at 24 hours; dashed portions indicate that fewer seeds remained active at the corresponding iteration count.

Reached/Completed discovers comparatively few states and plateaus early.

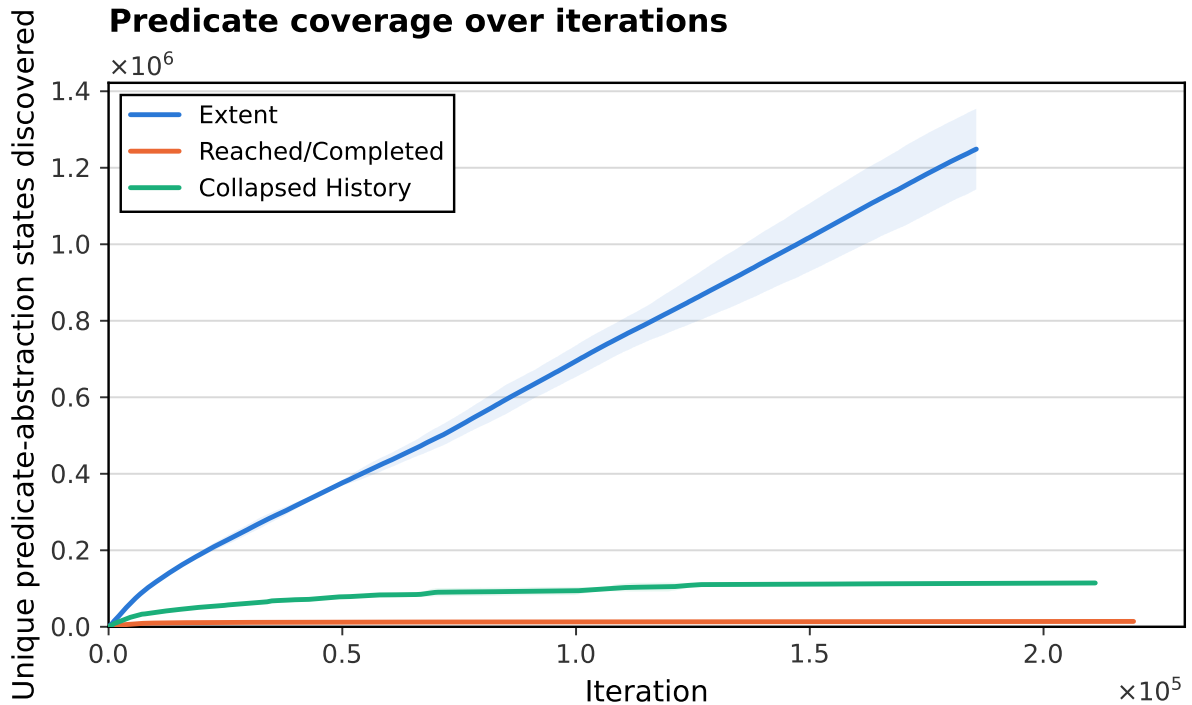


Figure 5.3: Coverage of the three predicate configurations measured using each configuration's own predicate-state abstraction. Extent retains substantially more distinctions and continues to discover new predicate states, whereas the two coarser configurations largely stagnate.

The effect of these plateaus is visible in Figure 5.4. Extent maintains a live mutation corpus throughout the measured budget and almost never needs to repopulate it. By contrast, Reached/Completed and Collapsed History repeatedly exhaust their queues once their native coverage signals stop identifying interesting schedules. Reached/Completed approaches one reseed per iteration over much of the run, while Collapsed History also reseeds frequently after its initial coverage growth slows. The approximately linear increase in their cumulative reseed counts shows that fresh executions are being introduced at a sustained rate rather than only occasionally. Since these repopulated executions are fresh unconstrained schedules, exploration beyond this point behaves increasingly like the Random baseline rather than sustained coverage-guided mutation.

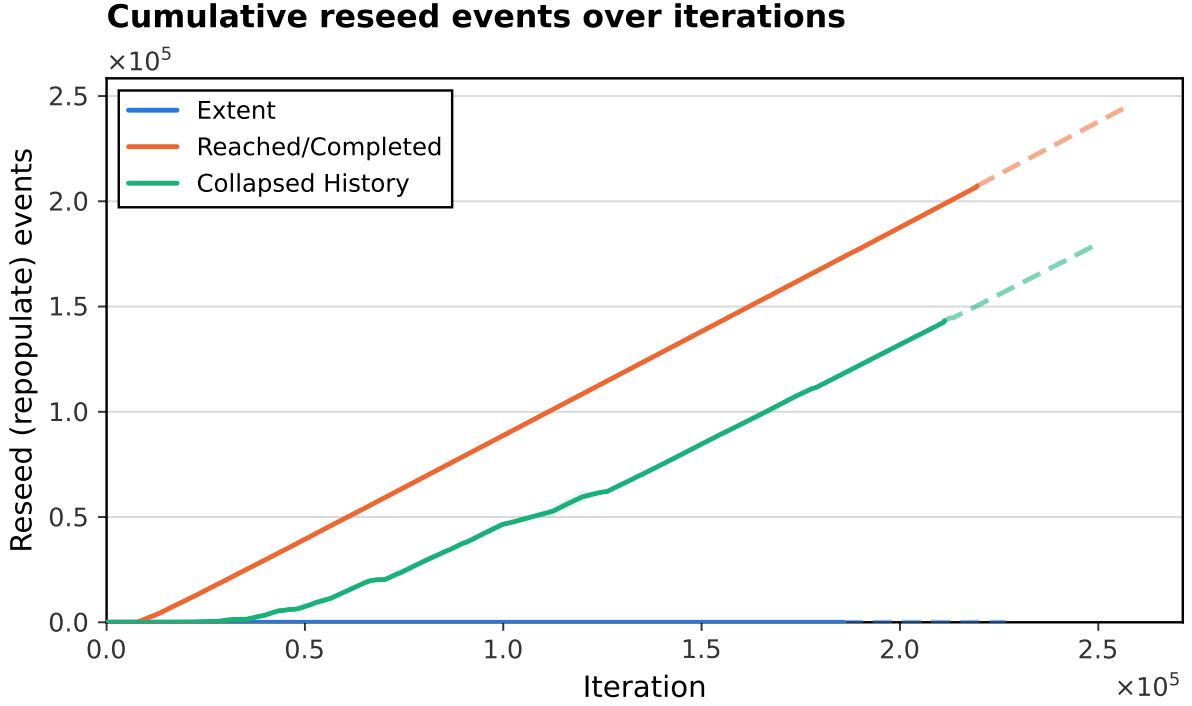


Figure 5.4: Cumulative queue-repopulation events for the three predicate configurations. Frequent reseeding indicates that the current corpus produced no schedules that were novel under the guiding predicate abstraction.

Stagnation in native predicate coverage does not mean that the executions stop reaching new states under every abstraction. Figure 5.5 evaluates the same runs using the common TLA+ state metric. The TLA+ coverage of Reached/Completed and Collapsed History continues to increase even after their own predicate signals have largely flattened. The Extent configuration still performs best, reaching approximately 1.2 million TLA+ states, but its advantage is much smaller than under its native metric. Its order-of-magnitude advantage in predicate states therefore does not translate proportionally into TLA+ state coverage.

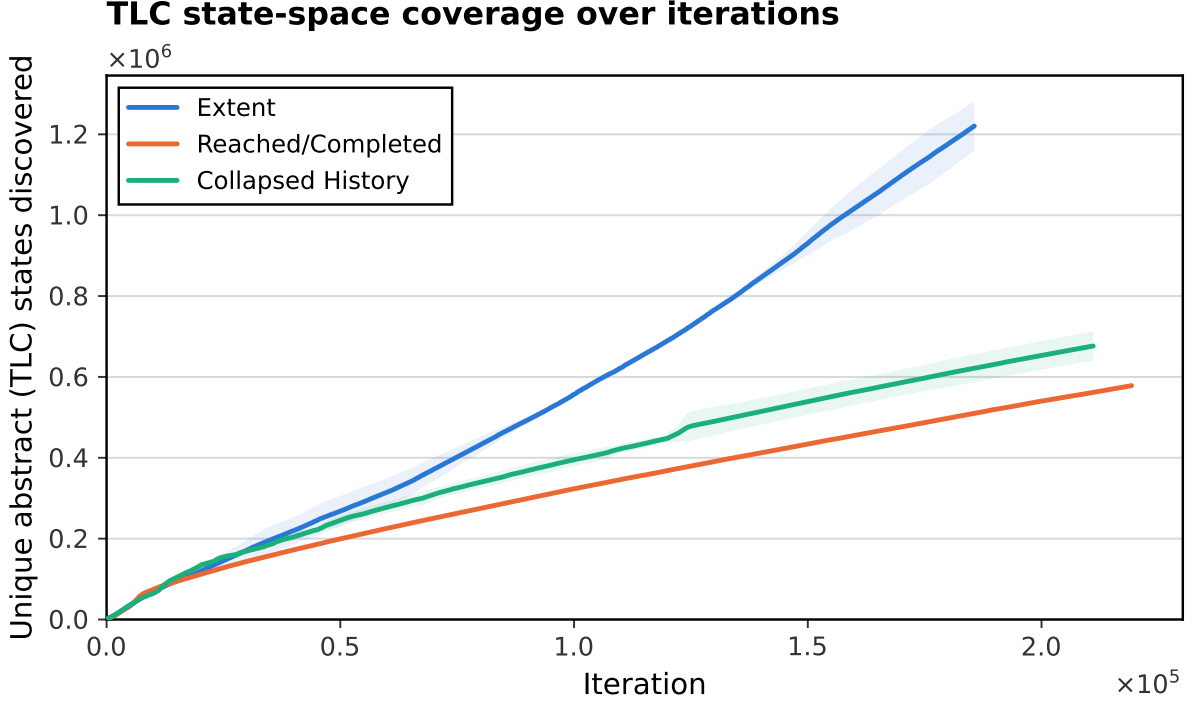


Figure 5.5: TLA+ state coverage reached by executions guided by the three predicate configurations. Unlike native predicate coverage, this common external metric continues growing for Reached/Completed and Collapsed History after their guidance signals stagnate.

Extent is the most detailed of the three abstractions, so one possible explanation for its advantage is that retaining more distinctions makes its feedback closer to trace-based guidance. The present results cannot determine whether its advantage comes from preserving particularly useful protocol information or simply from having a much larger state space. They do show that native coverage counts cannot be compared across predicate configurations as though they represented the same amount of protocol behaviour. None of the runs in this separate predicate-configuration experiment reproduced the invariant failure from Section 5.2.

5.5. Five- and Nine-Replica Configurations

The five- and nine-replica configurations test whether the conclusions from the main seven-replica experiment generalise at smaller and larger cluster sizes. At $n = 5$, the fast-path quorum matches the ordinary quorum. At $n = 9$, two replicas can be failed while enough replicas remain available for the Fast Path. Together, these configurations also show whether the relative ranking of the methods by TLA+ state coverage changes with cluster size.

The additional cluster sizes preserve the main coverage ordering. As shown in Figure 5.6, ModelFuzz discovers the most TLA+ states at both five and nine replicas, while Random discovers the fewest. Predicate guidance and Unguided Mutation remain closer together, and neither approaches ModelFuzz. The horizontal axes must be read separately. Within the wall-clock budget, the five-replica runs reach roughly 500,000 iterations, whereas the nine-replica runs reach only about 150,000. The lower nine-replica iteration count is consistent with its more expensive configuration, which uses more replicas, two injected faults, and schedules of up to 1000 events.

The fault-finding results in Figure 5.7 show a notable predicate anomaly at five replicas. Predicate guidance reaches the failure in three of four runs, compared with one of five runs at seven replicas and one of four at nine replicas. All three successful five-replica runs find it within approximately 42,000 iterations, while the sole nine-replica success occurs after approximately 61,000 iterations. The single seven-replica predicate success occurs earlier in iteration count, so the distinctive five-replica result is primarily its much higher success rate. Unguided Mutation remains the most reliable method overall,

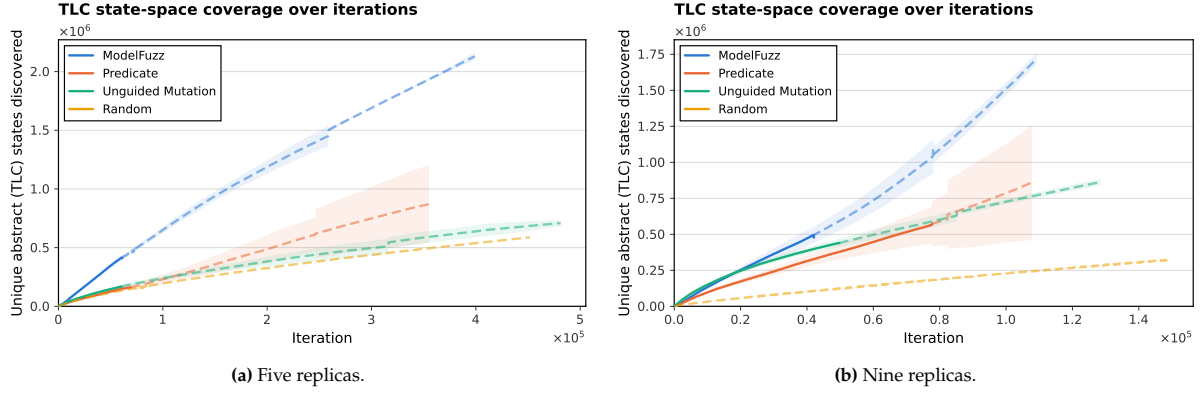


Figure 5.6: TLA+ state coverage for the five- and nine-replica configurations. The iteration axes differ because the runs complete different numbers of iterations within their wall-clock budgets.

while Random does not reach the failure at any cluster size.

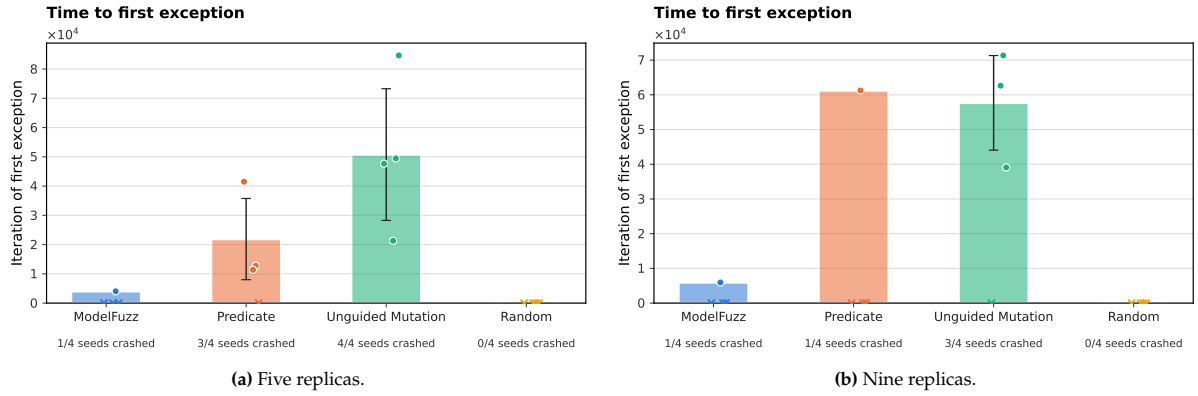


Figure 5.7: Iteration of the first observed exception in the five- and nine-replica configurations. Each configuration uses four runs per method. Error bars are calculated over successful runs only.

Across all three cluster sizes, the exception originates from the same line in `Command.validate`. The configurations therefore appear to exercise the same invariant failure rather than separate bugs, although the schedules that reach it differ.

5.6. Discussion

5.6.1. Summary of Findings

The clearest result is the advantage of ModelFuzz when all methods are evaluated using abstract TLA+ states. It reaches approximately 2.9 million states in the main comparison, compared with roughly 1.15 million for predicate guidance, 1.0 million for Unguided Mutation, and 550,000 for Random. This advantage is partly expected because ModelFuzz directly optimises novelty under the same abstraction used for evaluation. Nevertheless, the size and consistency of the difference show that model-state feedback is substantially more effective than the alternatives at exploring the state space represented by this TLA+ specification.

Predicate guidance performs less strongly but remains promising as a lightweight alternative. The Extent configuration exceeds both unguided baselines in TLA+ state coverage after approximately 100,000 iterations, showing that predicates can provide useful direction without executing the formal model. It still underperforms ModelFuzz by a wide margin, however. The results therefore support predicates as a potentially useful compromise between implementation cost and coverage, rather than as a replacement that matches model-guided exploration on the model's own metric.

The fault-finding comparison is less conclusive. Unguided Mutation reaches the observed invariant failure in all five runs, while both guided methods reach it only once. With five seeds and a single target failure, this establishes the outcome of these experiments but does not establish a general ordering of fault-finding effectiveness.

5.6.2. Crash-State Limitation

The TLA+ specification used by ModelFuzz does not represent which replicas are crashed. Executions that reach the same model variables while differing in the placement or duration of crashes can therefore collapse onto the same TLA+ state. This is particularly important for the discovered failure, whose original schedule depends on a crash during a narrow part of the PreAccept fanout. ModelFuzz receives no direct coverage reward for distinguishing such crash configurations, so its fault-finding result may underestimate what the algorithm could achieve with a model that represented failures explicitly.

The omission also limits the use of TLA+ coverage as a common external metric. Predicate states include the set of currently crashed replicas, as described in Section 3.2, and consequently distinguish executions that the TLA+ metric counts as identical. Predicate exploration may therefore be undercounted whenever it discovers new crash-dependent states that have no distinct representation in the model. The comparison remains valid as a measurement of coverage over this particular specification, but it is not a neutral measurement of every implementation state relevant to failures. This is a major limitation of both the cross-method coverage comparison and the interpretation of the fault-finding results.

5.6.3. Mapping Fidelity and State-Space Growth

The reachable state space observed by the framework grew substantially as the implementation-to-model mapping became more faithful. Mapping corrections can expose several model transitions for one implementation event. Recovery is an important example. The mapper now synthesizes `StartRecover` before applying the action corresponding to the first `Recover` delivery, as described in Section 4.3.3. Earlier versions that omitted one of these transitions skipped the intermediate states between them and consequently produced a much smaller coverage space.

This more detailed mapping is combined with schedules of up to 500 events and a TLA+ configuration that permits at least ten recovery attempts per replica. The recovery bound is necessary for the model to follow the Java execution throughout the schedule budget, but it also permits many more recovery interleavings. Figure 5.8 illustrates the resulting scale. A 48-hour ModelFuzz run exceeds eleven million unique TLA+ states after approximately 600,000 iterations and is still discovering new states at the end of the run. This differs from the earlier development behaviour, where the less complete mapper produced a visibly flatter coverage curve.

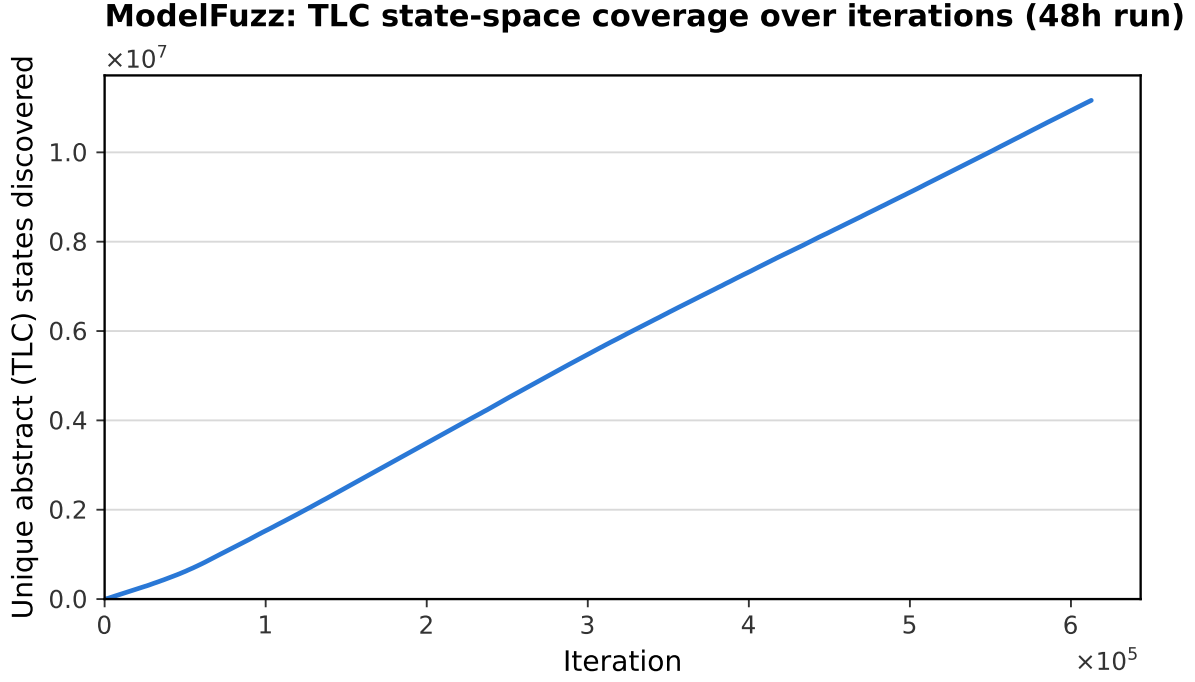


Figure 5.8: Unique abstract TLA+ states discovered during a 48-hour ModelFuzz run. Coverage continues to grow throughout the run under the corrected mapping and expanded recovery bound.

This growth prevents a meaningful test of saturation-triggered reseeding for the two strongest configurations. Neither ModelFuzz nor Extent exhausts its useful coverage signal within the available runs, so the experiments do not show how either would behave after reaching a plateau. Reseeding can be studied for Reached/Completed and Collapsed History precisely because their coarser abstractions saturate early. The absence of comparable reseeding observations for ModelFuzz and Extent should therefore not be interpreted as evidence that they never require reseeding.

5.6.4. Fault-Finding Results

The contrast between state coverage and fault finding shows that broader coverage does not necessarily improve the probability of finding every individual failure. Beyond the missing crash state discussed above, coverage-guided energy assignment may favour schedules that traverse many interesting states and produce longer descendants. The failure is initiated by a short eight-event trace, so a schedule that approaches it may appear uninteresting if those events add little coverage. Unguided Mutation does not impose this novelty filter and may therefore preserve simple mutations that guided exploration discards. The fact that Unguided Mutation reaches the exception while Random does not also suggests that mutation of existing schedules, rather than unguided generation alone, is important.

The current measurements do not establish this explanation. Testing it would require recording the length and novelty assigned to the ancestors of failing schedules and repeating the experiment with crash information represented by the model. Until such experiments are available, it should be read as a hypothesis rather than a conclusion.

5.6.5. Limitations

The CloudLab experiments run on a single machine, with one TLC server serving as many as four fuzzer instances concurrently. The instances therefore share processor time, memory bandwidth, and the TLC service rather than behaving as independent machines. This design increases experimental throughput, but resource contention may affect the number of iterations completed within a wall-clock budget. The resulting timing measurements should therefore be interpreted as system-level throughput under this configuration rather than as isolated per-fuzzer performance.

The experiments were also divided between CloudLab and a MacBook Air with an Apple M3 processor.

In particular, predicate-guided experiments and long-duration ModelFuzz experiments were run on the MacBook. Absolute wall-clock results from different environments are not directly comparable without accounting for their hardware and runtime differences. Comparisons should consequently use runs conducted in the same environment wherever possible.

Finally, the five- and nine-replica experiments were run with a different batch configuration from the main seven-replica experiment. They contain four runs per method rather than five, and their configuration produced different run lengths and reachable iteration counts. In particular, the nine-replica runs complete substantially fewer iterations within the same nominal wall-clock budget. These differences are experimental anomalies rather than controlled variables, so the cross-size results should be treated as qualitative checks and differences should not be attributed to cluster size alone.

The main averaged results are based on only five seeds, while the five- and nine-replica experiments use four. This is sufficient to expose the large difference in observed hit counts, but it is not sufficient for a strong statistical claim about fault-finding probability. Moreover, the time-to-exception statistics are conditioned on successful runs, leaving only one observation for both ModelFuzz and predicate guidance in the main comparison.

The recorded trace also has an instrumentation limitation. A message is recorded as a delivery when it is selected by the replay queue, before the crash simulator decides whether to drop it. The two PreAccept delivery attempts placed between the crash and recovery markers in the reproducing prefix may therefore be recorded as deliveries even when the crash gate drops them. The analysis above relies on observable replies and state transitions rather than interpreting every recorded delivery as observed.

6

Related Work

Testing a distributed protocol involves two related problems. A testing framework must control nondeterministic events such as message deliveries and faults, and it must then decide which of the resulting executions are worth exploring further. Prior work approaches the second problem through systematic state-space exploration, probabilistic schedule sampling, targeted fault injection, reinforcement learning, or coverage-guided fuzzing. This chapter positions the predicate-based approach within the wider field of software testing.

6.1. Systematic Schedule Exploration

Early implementation-level model checkers established controlled scheduling as a practical method for finding distributed concurrency bugs. **MaceMC** explores executions of systems written in the Mace language and identifies transitions that move the system into regions from which liveness cannot be recovered [12]. **MODIST** broadens this approach by transparently controlling the execution of unmodified distributed systems and systematically exploring message deliveries, process failures, and other nondeterministic choices [29]. These systems aim to enumerate a reduced execution space. The scheduler in this thesis provides similar control over deliveries and crashes, but uses sampled and mutated schedules rather than attempting exhaustive exploration.

Gulcan, Neto, and Ozkan’s **DSTest** is a general-purpose controlled-concurrency testing platform for distributed systems [9]. It intercepts communication at the application layer and exposes a modular interface through which testing algorithms can control message delivery and inject faults, without requiring language-specific instrumentation of the system under test. **DSTest** was considered as the scheduling substrate during the initial design of this work. We instead integrate directly into Accord’s **BurnTest** environment because its simulated network, logical time, internal tasks, and recovery timers share a single queue. Direct integration sacrifices some of **DSTest**’s generality, but gives the scheduler deterministic control over the internal progress on which Accord’s recovery behaviour depends.

Semantic reductions further restrict the schedules explored. **SAMC** uses developer-supplied rules to prune redundant reorderings involving messages and failures [15]. Drăgoi et al.’s communication-closure approach instead samples uniform lossy-synchronous executions, where processes advance in lockstep rounds and messages are delivered within their round or lost [8]. Both methods use protocol semantics to restrict the search space, whereas predicate guidance uses them to define coverage for unrestricted asynchronous schedules.

6.2. Probabilistic Schedule Sampling

Burckhardt et al.’s **Probabilistic Concurrency Testing** (PCT) characterises a bug by its depth, the number of scheduling constraints needed to expose it [4]. PCT assigns random priorities to threads and changes those priorities at randomly selected execution points, providing a lower bound on the probability of finding a bug of bounded depth. Although formulated for multithreaded programs, its depth-based view became the basis for probabilistic testing of distributed schedules.

Ozkan et al. generalise PCT to distributed systems with **PCTCP**, or PCT with Chain Partitioning [22]. PCTCP groups causally ordered events into chains as an execution unfolds, then uses random priorities to choose between concurrently available chains. This extends PCT’s probabilistic guarantees to distributed executions whose event ordering is not known in advance. Their later **taPCT** algorithm adds trace awareness, avoiding repeated exploration of schedules that differ only in the ordering of independent events [21]. These methods use a fixed random sampling strategy, whereas predicate guidance adapts using coverage accumulated across earlier executions.

6.3. Semantic and Feedback-Guided Exploration

Alvaro, Rosen, and Hellerstein’s **Molly** applies lineage-driven fault injection to distributed dataflow systems [1]. It reasons backwards from successful outcomes to derive combinations of omissions that could invalidate those outcomes, avoiding fault combinations that cannot affect correctness. Molly is therefore directed toward faults relevant to a particular result. Predicate guidance is not tied to a specified outcome or data lineage and instead rewards any new combination of protocol phases, quorum progress, and failures.

Mukherjee et al.’s **QL** applies reinforcement learning to controlled concurrency testing [18]. It learns a scheduling policy from the states and rewards observed during testing, allowing later scheduling choices to adapt to earlier executions. This establishes reinforcement learning as an alternative to independently sampled or systematically enumerated schedules. QL learns which scheduling actions are useful, whereas the fuzzer in this thesis retains explicit schedules and mutates those whose predicate or model states extend coverage.

Meng et al.’s **Mallory** adapts greybox fuzzing to distributed systems by constructing Lamport timelines from observed application and network events [16]. It abstracts these timelines into happens-before summaries and uses reinforcement learning to select sequences of faults that lead to previously unseen summaries. This makes Mallory an important example of semantic feedback derived from a running implementation. Its feedback requires the developer to select and expose relevant application events in addition to network traffic. In our setting, the feedback state is instead derived from the protocol messages already recorded by BurnTest, avoiding separate annotations inside Accord’s protocol logic.

Gulcan et al.’s **ModelFuzz** is the direct methodological baseline for this thesis [10]. It maps implementation events onto actions of an abstract formal specification and measures coverage over the model states reached by replaying those actions. This provides a protocol-aware abstraction that filters out many insignificant differences between schedules. Our implementation applies ModelFuzz to Accord and therefore retains its model-state coverage as a reference measure. The predicate-based alternative differs only in the source of feedback. Instead of replaying a TLA+ specification, it constructs states from transaction phases, quorum progress, and failures observed in the execution trace.

Borgarelli et al.’s **WaypointRL** uses reward augmentation to guide reinforcement-learning-based testing of distributed protocols [3]. The algorithm accepts an ordered sequence of developer-provided predicates describing semantic scenarios and rewards executions for reaching successive waypoints. This is the closest use of predicates to the approach developed in this thesis. Waypoint predicates express a particular target scenario and provide directed guidance toward it. Our predicates instead define general coverage over every observed transaction and phase, rewarding any newly reached combination without requiring a target execution to be specified in advance.

Zou et al.’s **DistFuzz** is a feedback-guided black-box fuzzer that jointly mutates regular events, faults, and timing intervals [30]. It uses network-message sequences as feedback and removes redundancies using order and role symmetry. Like predicate guidance, it derives feedback from communication rather than a formal model. DistFuzz retains message-sequence structure, whereas our abstraction summarises protocol phases and quorum progress.

The approach in this thesis lies between systematic, probabilistic, trace-based, and model-based guidance. Unlike **MaceMC**, **MODIST**, **SAMC**, and communication-closure testing, it does not attempt to systematically enumerate a reduced state space. Unlike **PCT**, **PCTCP**, and **taPCT**, it uses semantic feedback from previous executions rather than sampling each schedule according to a fixed probabilistic strategy. Unlike **Molly**, it is not directed toward faults that invalidate a particular outcome. Unlike **QL**

and **Mallory**, it does not learn a scheduling policy through reinforcement learning. Unlike **ModelFuzz**, it does not require an executable formal model or a mapping from every relevant implementation event to a specification action. Unlike **WaypointRL**, its predicates describe general protocol progress rather than a predetermined sequence of target states. Finally, unlike **DistFuzz**, it summarises executions in terms of protocol phases and quorum progress rather than network-message sequences.

The contribution is therefore not the general idea of semantic feedback, which appears in several forms throughout the literature. It is a lightweight coverage abstraction for phase-oriented distributed protocols, together with an implementation and evaluation that compare this abstraction directly against TLA+ model states and unguided random exploration on Apache Accord.

7

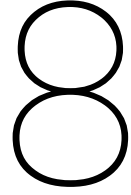
Future Work

The first priority is to represent process crashes and restarts explicitly in the TLA+ model. This would allow ModelFuzz to distinguish the crash-dependent states that matter to the discovered failure and would make TLA+ coverage a fairer external metric for predicate guidance. Repeating the experiments with this extension would show how much the missing failure state affected both coverage and fault-finding.

The evaluation should also be expanded across more seeds and protocol configurations. Further predicate variants can be constructed by varying the classification resolution and retained transaction history independently. Longer runs would help determine when the stronger Extent and ModelFuzz signals eventually saturate and how their queues behave once reseeding becomes necessary.

The invariant failure found in this work requires a more detailed root-cause analysis. Instrumenting local command-state transitions would identify where the invalid `executeAt` value is introduced and whether the same failure is present in earlier versions of Accord.

Finally, predicate guidance should be evaluated on other distributed protocols whose progress is organised into named stages or quorum-gated phases. This would test whether the abstraction is broadly useful or depends on properties specific to Accord's message vocabulary and recovery protocol.



Conclusion

This thesis investigated the use of abstract-state coverage to guide schedule exploration for Apache Accord. We implemented controlled execution inside Accord’s BurnTest framework, including message recording, schedule replay and mutation, process crashes, and a mapping from Java executions to actions in Accord’s TLA+ specification. We also introduced predicate coverage, which derives abstract states directly from transaction stages, quorum progress, and crashed replicas without executing a formal model.

Controlled exploration exposed a reproducible internal correctness failure in Accord. The failure violates an invariant requiring a command’s `executeAt` timestamp to follow its transaction identifier. Its execution involves partial `PreAccept` delivery, a coordinator crash, and overlapping recovery activity. The same assertion is reached in the experiments with five, seven, and nine replicas, indicating that they exercise the same failure mechanism. The available trace does not establish a client-visible linearizability violation or identify the exact local state transition that introduces the invalid timestamp, but it demonstrates that controlled schedule exploration can reveal faults in Accord’s production protocol logic.

The clearest experimental result concerns abstract TLA+ state coverage. ModelFuzz consistently discovers substantially more model states than predicate guidance and both unguided baselines. This is expected in part because ModelFuzz directly optimises the metric used for evaluation, but the size of the difference confirms that model states provide highly effective guidance when a sufficiently faithful mapping is available. Predicate guidance does not match ModelFuzz, although its `Extent` configuration exceeds the unguided baselines in the main comparison. The coarser predicate configurations lose useful distinctions, saturate early, and increasingly depend on fresh random schedules. Predicate coverage therefore offers useful semantic guidance at a lower implementation cost, but its effectiveness depends strongly on the resolution of the abstraction.

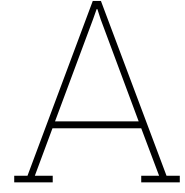
The fault-finding results are less decisive. Unguided Mutation reaches the discovered failure most reliably in the seven-replica experiments, while predicate guidance performs well with five replicas. Random exploration does not reach the failure in any configuration. These results show that broader state coverage does not guarantee a higher probability of finding a particular fault. They do not establish a general ranking of the methods because the experiments use few seeds and the TLA+ model does not represent crashed replicas. This omission both prevents ModelFuzz from receiving direct guidance toward crash-dependent executions and causes the common TLA+ metric to merge states that predicate coverage distinguishes.

Overall, ModelFuzz is the strongest method for exploring the behaviour represented by Accord’s TLA+ specification. Predicate coverage is a practical alternative when maintaining a formal model and its execution mapping is too costly, and it can improve upon unguided exploration while remaining closely tied to protocol progress. The implementation developed in this thesis provides a basis for further investigation of the discovered failure, explicit crash modelling, and predicate guidance in other phase-oriented distributed protocols.

References

- [1] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. “Lineage-Driven Fault Injection”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2015, pp. 331–346. doi: 10.1145/2723372.2723711.
- [2] Peter Bailis and Kyle Kingsbury. “The network is reliable”. In: *Commun. ACM* 57.9 (Sept. 2014), pp. 48–55. ISSN: 0001-0782. doi: 10.1145/2643130. URL: <https://doi.org/10.1145/2643130>.
- [3] Andrea Borgarelli et al. “Reward Augmentation in Reinforcement Learning for Testing Distributed Systems”. In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA2 (2024). doi: 10.1145/3689779.
- [4] Sebastian Burckhardt et al. “A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs”. In: *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2010, pp. 167–178. doi: 10.1145/1736020.1736040.
- [5] Mike Burrows. “The Chubby Lock Service for Loosely-Coupled Distributed Systems”. In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. Seattle, WA: USENIX Association, Nov. 2006, pp. 335–350. URL: <https://www.usenix.org/conference/osdi-06/chubby-lock-service-loosely-coupled-distributed-systems>.
- [6] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. Seattle, WA: USENIX Association, Nov. 2006, pp. 205–218. URL: <https://www.usenix.org/conference/osdi-06/bigtable-distributed-storage-system-structured-data>.
- [7] Giuseppe DeCandia et al. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP ’07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 205–220. doi: 10.1145/1294261.1294281.
- [8] Cezara Dragoi et al. “Testing Consensus Implementations Using Communication Closure”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020). doi: 10.1145/3428278.
- [9] Ege Berkay Gulcan, João Neto, and Burcu Kulahcioglu Ozkan. “Generalized Concurrency Testing Tool for Distributed Systems”. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, pp. 1861–1865. ISBN: 9798400706127. doi: 10.1145/3650212.3685309. URL: <https://doi.org/10.1145/3650212.3685309>.
- [10] Ege Berkay Gulcan et al. “Model-Guided Fuzzing of Distributed Systems”. In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA2 (2025). doi: 10.1145/3763060.
- [11] Patrick Hunt et al. “ZooKeeper: Wait-Free Coordination for Internet-Scale Systems”. In: *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010, pp. 145–158. URL: <https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems>.
- [12] Charles Edwin Killian et al. “Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code”. In: *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 07)*. Cambridge, MA: USENIX Association, Apr. 2007. URL: <https://www.usenix.org/conference/nsdi-07/life-death-and-critical-transition-finding-liveness-bugs-systems-code>.
- [13] Avinash Lakshman and Prashant Malik. “Cassandra: a decentralized structured storage system”. In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. doi: 10.1145/1773912.1773922. URL: <https://doi.org/10.1145/1773912.1773922>.

- [14] Leslie Lamport. “The Part-Time Parliament”. In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169. doi: 10.1145/279227.279229.
- [15] Tanakorn Leesatapornwongsa et al. “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)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 399–414. ISBN: 978-1-931971-16-4.
- [16] Ruijie Meng et al. “Greybox Fuzzing of Distributed Systems”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2023, pp. 1615–1629. doi: 10.1145/3576915.3623097.
- [17] Iulian Moraru, David G. Andersen, and Michael Kaminsky. “There is more consensus in Egalitarian parliaments”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. Farmington, Pennsylvania: Association for Computing Machinery, 2013, pp. 358–372. ISBN: 9781450323888. doi: 10.1145/2517349.2517350. URL: <https://doi.org/10.1145/2517349.2517350>.
- [18] Suvam Mukherjee et al. “Learning-Based Controlled Concurrency Testing”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020). doi: 10.1145/3428298.
- [19] Brian M. Oki and Barbara H. Liskov. “Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems”. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. PODC ’88. New York, NY, USA: Association for Computing Machinery, 1988, pp. 8–17. doi: 10.1145/62546.62549.
- [20] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2.
- [21] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. “Trace Aware Random Testing for Distributed Systems”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019). doi: 10.1145/3360606.
- [22] Burcu Kulahcioglu Ozkan et al. “Randomized Testing of Distributed Systems with Probabilistic Guarantees”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018). doi: 10.1145/3276530.
- [23] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. *A problematic run in Apache Accord*. Tech. rep. IMDEA Software Institute and Telecom SudParis, 2025.
- [24] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. *Making Democracy Work: Fixing and Simplifying Egalitarian Paxos (Extended Version)*. 2025. arXiv: 2511.02743 [cs.DC]. URL: <https://arxiv.org/abs/2511.02743>.
- [25] Fred B. Schneider. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”. In: *ACM Computing Surveys* 22.4 (Dec. 1990), pp. 299–319. ISSN: 0360-0300. doi: 10.1145/98163.98167.
- [26] Benedict Elliott Smith et al. “Accord: Fast Geo-Distributed Transactions in Apache Cassandra”. Unpublished manuscript. 2026.
- [27] Benedict Elliott Smith et al. *Draft Whitepaper for CEP-15: General Purpose Transactions*. White paper. Apple Inc. and University of Michigan, 2023.
- [28] Pierre Sutra. *On the correctness of Egalitarian Paxos*. 2019. arXiv: 1906.10917 [cs.DC]. URL: <https://arxiv.org/abs/1906.10917>.
- [29] Junfeng Yang et al. “MODIST: Transparent Model Checking of Unmodified Distributed Systems”. In: *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*. Boston, MA: USENIX Association, Apr. 2009, pp. 213–228. URL: <https://www.usenix.org/conference/nsdi-09/modist-transparent-model-checking-unmodified-distributed-systems>.
- [30] Yonghao Zou et al. “Blackbox Fuzzing of Distributed Systems with Multi-Dimensional Inputs and Symmetry-Based Feedback Pruning”. In: *Proceedings of the 32nd Network and Distributed System Security Symposium (NDSS)*. 2025. doi: 10.14722/ndss.2025.241912.



Trace for Iteration 58,372

The invariant failure discussed in Section 5.2 was discovered at fuzzer iteration 58,372 in a seven-replica execution. The iteration number is the fuzzer loop counter rather than a random seed. The failing mutant was derived from seed -7610621359446545149 and has been reduced to an eight-event controlled prefix. Once this prefix is exhausted, BurnTest continues processing the messages and internal tasks produced by the protocol. The resulting trace contains 214 numbered events, from event 0 through event 213.

Table A.1 shows the controlled prefix. It contains only the crash and recovery of n_2 that affect the replay. Five Recover actions for nodes n_7 , n_4 , and n_3 were removed from the earlier 13-event reproducer because none of those nodes had been marked as crashed. The shorter prefix reproduces the same exception and recovery pattern.

The replay queue records a Deliver action before the crash gate is evaluated. A delivery selected while n_2 is marked as crashed is therefore an attempted delivery rather than proof that the packet reached its destination. This distinction is important for interpreting the two PreAccept attempts between events 1 and 4.

Event	Controlled action	Observed role in the execution
0	Deliver client $\rightarrow n_2$	Submit the first transaction to coordinator n_2 .
1	Crash n_2	Enable the crash gate for messages sent from or to n_2 .
2	Deliver $n_2 \rightarrow n_6$	Attempt PreAccept delivery while the crash marker is active.
3	Deliver $n_2 \rightarrow n_1$	Attempt PreAccept delivery while the crash marker is active.
4	Recover n_2	Clear the crash marker for the coordinator.
5	Deliver $n_2 \rightarrow n_5$	Deliver PreAccept after recovery.
6	Deliver $n_2 \rightarrow n_4$	Deliver PreAccept after recovery.
7	Deliver $n_2 \rightarrow n_3$	Deliver PreAccept after recovery.

Table A.1: The minimized eight-event controlled prefix used to reproduce the iteration 58,372 failure.

Table A.2 condenses the relevant portions of the full recorded execution. It omits routine status messages and replies that do not change the interpretation of the failure.

Recorded events	Interpretation
0–13	The first transaction is submitted to n_2 and its PreAccept round reaches only part of the cluster before the crash marker is cleared.
14–77	Two further transactions progress after recovery, providing a control period after the initial partial delivery.
78–148	Accord’s progress mechanism exchanges status messages for transactions that appear stuck.
149–154	Node n_4 begins recovery of the first transaction.
159–161, 168, 172, 187	Node n_4 collects RecoverOK responses for the first transaction.
175–180	Node n_2 begins recovery of a later transaction.
181–186	Node n_5 independently begins a competing recovery of that later transaction.
188–197	Nodes n_2 and n_5 collect responses for their concurrent recovery attempts.
199–204	The recovery led by n_4 broadcasts Accept on the Slow Path for the first transaction.
208–213	Node n_4 also begins recovery of the later transaction.
After event 213	Applying one of events 199–204 raises the <code>executeAt</code> ordering invariant in <code>Command.validate</code> .

Table A.2: Condensed phases of the 214-event execution produced from the minimized controlled prefix.

The trace establishes that partial PreAccept delivery is followed by overlapping recovery attempts and then by the failing Accept exchange. It does not reveal which replica throws the exception or the precise local command-state updates responsible for the invalid timestamp. Those details require per-replica command-store instrumentation and remain open.