Multi-Grained Specifications for Distributed System Model Checking and Verification

Lingzhi Ouyang
SKL for Novel Soft. Tech.
Nanjing University, China
lingzhi.ouyang@smail.nju.edu.cn

Xudong Sun
University of Illinois
Urbana-Champaign, IL, USA
xudongs3@illinois.edu

Ruize Tang
SKL for Novel Soft. Tech.
Nanjing University, China
tangruize@smail.nju.edu.cn

Yu Huang*
SKL for Novel Soft. Tech.
Nanjing University, China
yuhuang@nju.edu.cn

Madhav Jivrajani
University of Illinois
Urbana-Champaign, IL, USA
madhavj2@illinois.edu

Xiaoxing Ma
SKL for Novel Soft. Tech.
Nanjing University, China
xxm@nju.edu.cn

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

Abstract

This paper presents our experience specifying and verifying the correctness of ZooKeeper, a complex and evolving distributed coordination system. We use TLA$^+$ to model fine-grained behaviors of ZooKeeper and use the TLC model checker to verify its correctness properties; we also check conformance between the model and code. The fundamental challenge is to balance the granularity of specifications and the scalability of model checking—fine-grained specifications lead to state-space explosion, while coarse-grained specifications introduce model-code gaps. To address this challenge, we write specifications with different granularities for composable modules, and compose them into mixed-grained specifications based on specific scenarios. For example, to verify code changes, we compose fine-grained specifications of changed modules and coarse-grained specifications that abstract away details of unchanged code with preserved interactions. We show that writing multi-grained specifications is a viable practice and can cope with model-code gaps without untenable state space, especially for evolving software where changes are typically local and incremental. We detected six severe bugs that violate five types of invariants and verified their code fixes; the fixes have been merged to ZooKeeper. We also improve the protocol design to make it easy to implement correctly.

CCS Concepts: • **Software and its engineering → Model checking**; • **Computer systems organization → Reliability**.

Keywords: Distributed systems, model checking, reliability

ACM Reference Format:
Lingzhi Ouyang, Xudong Sun, Ruize Tang, Yu Huang, Madhav Jivrajani, Xiaoxing Ma, and Tianyin Xu. 2025. Multi-Grained Specifications for Distributed System Model Checking and Verification. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3689031.3696069

1 Introduction

Distributed systems that implement complex protocols are notoriously difficult to develop and maintain. It is non-trivial to implement non-deterministic, asynchronous behavior and reason about correctness and fault tolerance. Formal methods have been increasingly used to verify protocol designs [53, 77, 78, 80, 84, 102–104] and system code [55, 57, 88, 89, 98]. Recently, formal methods start to go beyond academic research towards validating and verifying production distributed systems [32, 35, 52, 81]. For example, companies like Amazon, Azure, MongoDB, and LinkedIn all use TLA$^+$ to specify and model-check production systems [31, 36, 65, 74, 106].

This paper presents our experience specifying and verifying ZooKeeper [58], a complex, evolving distributed coordination system which uses a totally ordered broadcast protocol, known as Zab [60, 61, 85]. ZooKeeper is actively maintained as an open-source project [17]; it is widely used in practice as a critical infrastructure system for storing service metadata and for fault tolerance [27, 39, 56, 93]. We aim

*Corresponding author

EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696069

to verify not only the Zab protocol, but also the implementation in ZooKeeper. We use TLA$^+$ to specify ZooKeeper's behavior and use the TLC model checker to verify that every execution satisfies all correctness properties [25, 66]. To ensure that our model correctly describes the implementation's behavior, we run conformance checking to detect discrepancies between the model and the implementation (§3.4). Note that we target existing ZooKeeper code written in Java, mostly maintained by developers with little formal method expertise; so, rewriting ZooKeeper with machine-checked proofs in verification-aware languages [43, 67, 71] or frameworks [57, 89] is not realistic in the short term.

The fundamental challenge is to balance the scalability of model checking and the granularity of specifications in modeling code-level implementation. On one hand, fine-grained specifications, which model code-level behavior, lead to *state-space explosion*. As a data point, model checking using TLC on ZooKeeper's official system specification in TLA$^+$ cannot finish in ten days with a standard configuration of three nodes, three transactions, up to three node crashes, and up to three network partitions [79]. In fact, this TLA$^+$ specification omits several important code behavior (§2.1.2) like multithreading concurrency; modeling those would be even more costly. Unfortunately, as subtle bugs often reside in deep states, extensive state-space exploration is inevitable.

On the other hand, coarse-grained specifications introduce *model-code gaps*—the specification does not effectively reflect code-level implementation; consequently, verification or model checking cannot capture subtle bugs whose manifestations are abstracted away from the model. We find that model-code gaps are prevalent. One main reason is that implementations are highly optimized and the optimizations are rarely modeled in existing specifications. A typical case is that an atomic action in the Zab protocol is implemented by several concurrent operations in ZooKeeper (for performance optimization). If the specification only models the protocol, bugs manifested via interleavings of concurrent operations cannot be exposed. Such gaps are common in distributed systems projects: we inspected TLA$^+$ specifications of MongoDB [21], CCF [20], TiDB [10], etcd [24], and CosmosDB [9]; local concurrency is often abstracted away.

To address this challenge, we write *multi-grained specifications*, i.e., multiple specifications with different granularities for composable modules, and compose them into *mixed-grained specifications* for specific scenarios. For example, to verify a code change, we compose a mixed-grained specification using fine-grained specifications of changed modules and coarse-grained specifications that abstract away details of unchanged code. Essentially, this approach allows model checkers to focus on the target modules with fine-grained modeling that reflects the implementation. To enable multi-grained specifications, we write composable specifications for each module with an interaction-preserving principle, where a coarse-grained specification coarsens the corresponding fine-grained specification while preserving all actions whose effects are visible to the other modules.

To divide the specification into easily composable modules, we leverage an opportunity that Zab, like other distributed protocols (e.g., Paxos, Raft, and 2PC), is designed to run in phases, with clean boundaries between phases. For example, Zab runs in four phases (Election, Discovery, Synchronization and Broadcast) sequentially if no failure happens. Thus, we decompose the ZooKeeper specification by phases and write multi-grained specifications for each phase.

We show that multi-grained specification is a viable practice and can effectively address model-code gaps without untenable state-space explosion, especially for evolving software where changes are typically local and incremental. We are able to model low-level system behavior such as local concurrency in fine-grained specifications and use them to create mixed-grained specifications with manageable state space. This practice allows us to capture deep bugs that cannot be found with existing TLA$^+$ specifications; it also allows us to efficiently verify code changes and bug fixes, which can introduce new bugs or fail to resolve the root cause. The efforts of writing multiple specifications are manageable and are done incrementally. As many distributed system projects have already adopted the practice of writing TLA$^+$ specifications, we demonstrate the methodology to deepen TLA$^+$ specifications to verify system implementations.

We develop a framework for model checking and verification of distributed systems with multi-grained specifications, named Remix. It composes module specifications into mixed-grained specifications. It also provides conformance checking to preclude deviations that could be introduced when writing new specifications. Using Remix, we have detected six deep bugs in ZooKeeper code and verified their fixes. The effort also helps improve the Zab protocol to make it easy to implement correctly. Our evaluation shows that mixed-grained specifications can significantly outperform existing specifications in verification effectiveness and efficiency.

This paper makes the following main contributions:

- We share our practice of writing multi-grained specifications with the interaction preserving principle;
- We present our mechanism and tooling for composing multi-grained specifications of different modules into the mixed-grained specification;
- We demonstrate the values of fine-grained modeling that reconciles specifications and code implementation.
- We found six deep bugs in ZooKeeper, verified their code fixes, and improved the protocol design.
- Our artifact: https://zenodo.org/records/13738672.

2 Background

2.1 Existing Specifications

We started by writing TLA$^+$ specifications for the Zab protocol and the ZooKeeper system (its realization of the Zab

Phase 2 (Synchronization)

...

Step $f.2.1$ Upon receiving the $NEWLEADER(e', T)$ message from l,
the follower starts a new iteration if $f.p \neq e'$.
If $f.p = e'$, then it executes the following actions atomically:
① 1) It sets $f.a$ to e';
② 2) For each $\langle v, z \rangle \in E(I_{e'})$, it accepts $\langle e', \langle v, z \rangle \rangle$, and make $h_f = T$.
③ Finally, it acknowledges the $NEWLEADER(e', I_{e'})$ proposal to the leader, thus accepting the transactions in T.

(a) Pen-and-paper description in the Zab paper [61]

```
    FollowerProcessNEWLEADER(i, j) ==
      /\ IsFollower(i) /\ IsMyLeader(i, j)
      /\ PendingNEWLEADER(i, j)
      /\ LET msg == msgs[j][i][1]
             epochOk == acceptedEpoch[i] = msg.mepoch
         IN \/ /\ ~epochOk
               /\ FollowerShutdown(i) /\ ...
            \/ /\ epochOk /\ ...
               /\ currentEpoch' = [currentEpoch EXCEPT
①                                        ![i] = acceptedEpoch[i]]
②              /\ history' = [history EXCEPT ![i] = msg.mhistory]
               /\ LET m == [mtype |-> ACKLD,
③                           mzxid |-> LastZxidOfHistory(history'[i])]
                  IN Reply(i, j, m)
               /\ UNCHANGED <<state, zabState, ...>>
```

(b) Protocol specification in TLA$^+$

Figure 1. Pen-and-paper description and TLA$^+$ specification of Step $f.2.1$ in Phase 2 of the Zab protocol.

```
    while (self.isRunning()) {
      readPacket(qp);
      switch (qp.getType()) { ...
        case Leader.NEWLEADER:
          ...
①         self.setCurrentEpoch(newEpoch);
          zk.startupWithoutServing();
          if (zk instanceof FollowerZooKeeperServer) {
            FollowerZooKeeperServer fzk = zk;
②           for (PacketInFlight p : packetsNotCommitted)
              fzk.logRequest(p.hdr, p.rec, p.digest);
            packetsNotCommitted.clear();
          }
          writePacket(
③           new QuorumPacket(Leader.ACK, newLeaderZxid, ...));
          break;
    }} // zookeeper-server/src/.../server/quorum/Learner.java
```

(a) Code implementation (v3.9.1) in Java [28]

```
    FollowerProcessNEWLEADER(i, j) ==
      /\ IsON(i) /\ IsFollower(i) /\ IsMyLeader(i, j)
      /\ PendingNEWLEADER(i, j)
      /\ LET packetsInSync == packetsSync[i].notCommitted
             ms_ack == ACKInBatches(<< >>, packetsInSync)
             msg == msgs[j][i][1]
             m_ackld == [mtype |-> ACKLD, mzxid |-> msg.mzxid]
             queue_toSend == <<m_ackld>> \o ms_ack
         IN /\ ...
①           /\ currentEpoch' = [currentEpoch EXCEPT
                                  ![i] = acceptedEpoch[i]]
            /\ history'      = [history EXCEPT
②                                ![i] = @ \o packetsInSync]
            /\ packetsSync'  = [packetsSync EXCEPT
                                 ![i].notCommitted = << >>]
②③          /\ Reply(i, j, queue_toSend)
      /\ UNCHANGED <<state, acceptedEpoch, ...>>
```

(b) System specification in TLA$^+$

Figure 2. Code implementation in Java and the corresponding system specification in TLA$^+$ of Step $f.2.1$ in Phase 2 of the Zab protocol in ZooKeeper.

protocol), respectively. Both are now part of the official TLA$^+$ specifications of the ZooKeeper projects.

2.1.1 Protocol specification.

The protocol specification follows the pen-and-paper description of the original Zab paper [61]. The goal is to formally describe and model check the protocol. Figure 1 shows the snippet of the protocol specification of Step $f.2.1$ in Phase 2 (Synchronization) of Zab [61], where the follower is supposed to *atomically* execute two actions upon receiving a NEWLEADER message from the leader: ① updating its current epoch and ② accepting the leader's complete history. In the protocol specification, we write the actions, together with the final acknowledgment, in a TLA$^+$ atomic action FollowerProcessNEWLEADER.

Our TLA$^+$ specifications also specify several missing components that are not described in the Zab protocol, e.g., the Zab protocol does not describe leader election (it uses an assumed leader oracle) and does not describe certain unexpected cases (e.g., when the leader does not receive sufficient acknowledgments from the followers).

2.1.2 System specification.

With the protocol specification, we then wrote the system specification of ZooKeeper, as a precise, testable system design document [81]. The system specification is developed based on the ZooKeeper source code, instead of the Zab paper. For example, ZooKeeper implements a fast leader election algorithm, which is specified in the system specification, which refines the leader oracle in the protocol specification.

Figure 2a shows the code snippet of ZooKeeper's implementation of Step $f.2.1$ of the Zab protocol. When a follower receives the NEWLEADER message, it ① updates the current epoch, ② logs every packet that is not committed, and ③ replies to the leader. Figure 2b shows the corresponding system specification. Note that the system specification does not strictly refine the protocol specification. The system implementation optimizes the synchronization using NEWLEADER as a signaling message without carrying concrete history, and the leader's history will be synchronized in one of three modes (DIFF, TRUNC, and SNAP), depending on the differences between the history and the follower's latest transaction ID (zxid). In this paper, since our goal is to verify the ZooKeeper system implementation, we start with the system specification instead of the protocol specification.

2.2 Model-Code Gaps

Despite that the system specification effectively describes the ZooKeeper system, it still omits certain implementation details. We present three common patterns of model-code gaps which in our experience is important to consider as they often induce tricky and error-prone implementation. Overlooking them would allow bugs to escape from model checking and reduce the confidence of verification results.

2.2.1 Atomicity. As a common model-code gap pattern, an atomic action in the system specification is not guaranteed to be atomically executed at the code level. In this case, if a specified atomic action is partially executed and then interrupted, the intermediate states would be missed in model checking. In Figure 2b, `FollowerProcessNEWLEADER` is an atomic action: the state transitions from ① to ② to ③ are always atomically done in the system specification. However, ZooKeeper's code-level execution does not guarantee such atomicity (Figure 2a). We will show in §5 that model-code gaps due to false atomicity would miss critical bugs.

We find that atomicity-related model-code gaps are common in TLA$^+$ specifications of many distributed systems. One reason is that every action in TLA$^+$ is atomic so it is convenient to express logically connected steps in an action.

2.2.2 Concurrency. We find that specifications commonly focus on non-deterministic interleavings of actions *among* nodes, aka distributed concurrency [70]. Few model local concurrency within a node (e.g., due to multithreading). Instead, the specification often models locally concurrent events in a single action with deterministic orders. However, if multithreaded code has non-deterministic behavior, model checking using such specifications would fail to explore bug-triggering states. For example, in Figure 2a, the follower's `QuorumPeer` thread calls `logRequest()` (line 11). The implementation of `logRequest` sends a logging request, which would be asynchronously handled by a different thread. However, the above procedure is simplified as the state transition of appending all uncommitted requests to the follower's history (line 12-13 in Figure 2b), with sending replies to the leader in a deterministic order (line 16 in Figure 2b). Consequently, checking the specification will miss many possible states of asynchronous logging (§5).

Similar to atomicity, we find that model-code gaps related to local concurrency are common in existing TLA$^+$ specifications of many other distributed systems projects.

2.2.3 Missing state transitions. State transitions in the specification may be overly simplified compared to the code implementation. For example, the follower in ZooKeeper would reply `ACK` upon receiving an `UPTODATE` message; in the specification, the follower does not reply `ACK` to `UPTODATE` for simplicity. Missing state transitions can cause model checking to miss possible states, and meanwhile, explore false states that cannot be reached by code-level executions.

2.3 Challenges

The prevalence of model-code gaps in existing specifications indicates the need to further model important, fine-grained behavior like non-atomic updates and concurrency for verifying code implementation (which is uncommon in existing TLA$^+$ specifications). However, doing so would significantly increase state space, resulting in state-space explosion. Currently, using TLC to model check the system specification of ZooKeeper (§2.1.2) cannot finish in ten days with a standard configuration (three nodes, three transactions, up to three node crashes, and up to three network partitions) [79]. How to balance the granularity of the specification and the scalability of model checking is a key challenge.

3 Writing Multi-Grained Specifications

We write *multi-grained specifications*, i.e., multiple specifications with different granularities for composable modules, which can be composed into *mixed-grained specifications* with preserved interactions. A mixed-grained specification consists of *fine-grained* specifications of target modules to model code-level behavior and *coarse-grained* specifications of other modules to save cost. Mixed-grained specifications enable us to verify the system module by module [29, 30, 41, 59], and to verify code changes or bug fixes.

We write our specifications in TLA$^+$ which offers inherent flexibility to choose and adjust the abstraction level. We present the principles of writing multi-grained specifications with composability (§3.1–§3.3). We use conformance checking (§3.4) to match specifications with code implementation.

Concretely, use cases of multi-grained specifications are:

- **Verifying protocol designs.** We verify the Zab algorithm using the protocol specification (§2.1.1). As the protocol specification models high-level algorithms, it is verified in a traditional way without mixed-grained specifications.
- **Verifying system designs.** As discussed in §2.1.2, the system specification could take a long time to check, especially with complex configurations. Mixed-grained specifications help the model checker speed up the verification.
- **Verifying system implementations.** Mixed-grained specifications enable fine-grained modeling of code behavior to verify the implementation. The cost of model checking is managed by coarsening the modules that are not verification targets. Conformance checking is needed to ensure specifications conform to code implementations.
- **Verifying code changes.** Mixed-grained specification also allows efficient verification of code changes (e.g., bug fixes). As code changes are typically local and incremental, we can use fine-grained specifications for the changed modules while coarsening the unchanged ones.

3.1 Fine-Grained Specifications

For a given specification, we write its fine-grained counterpart by modeling low-level code behavior. For a target action in the original specification, we rewrite the enabling conditions based on code logic and next-state updates of the action. We focus on three patterns of model-code gaps (§2.2):

- **Atomicity.** We rewrite an action that is not guaranteed to be atomically executed at the code level. We split the action into multiple separate actions in the specification and set up their enabling conditions accordingly.
- **Concurrency.** To model concurrency, we separate state transitions, which are executed by different threads, into different actions as per the executing threads. Inter-thread communications (e.g., message queues for local thread messages) are also modeled in the specification.
- **Missing state transitions.** We focus on enhancing state transitions with existing variables in the target action. If the enabling condition includes other dependent variables that are missed, we add them to the specification.

We decide atomic blocks based on how threads/nodes communicate, following prior work [55]. An atomic block starts with reading the external state, performing internal computations, and ends with writing to the external state. The results of internal computations are invisible to other threads/nodes, thus can be safely folded in an atomic block. An example atomic block starts with receiving a message from the network, and ends with putting another message in a queue to be handled by another thread.

Case study: Fine-grained modeling of Step $f.2.1$ in Figure 2b. We rewrite the system specification to model non-atomic actions and local concurrency with fine-grained specifications. We first rewrite the atomic `FollowerProcessNEWLEADER` action into three actions corresponding to steps ①, ②, and ③ in Figure 2b, as there is no atomicity guarantee at the code level. The fine-grained model allows model checkers to explore intermediate states in non-atomic executions. Figure 3 shows the three actions in the refined TLA$^+$ specification.

To set up correct triggering of these fine-grained actions, we specify their enabling conditions using existing or new variables based on the code implementation. For example, action `FollowerProcessNEWLEADER_LogAsync` models the logic of queuing requests for asynchronous logging (Figure 3b). It is enabled when the follower has updated its `currentEpoch` (line 4) and there exist packets to be logged (line 5), which corresponds to the conditions in the code (Figure 2a).

We then rewrite the `FollowerProcessNEWLEADER_LogAsync` action for concurrency by specifying the asynchronous logging logic. To do so, we decouple the logging action by a separate thread from the follower's message-handling actions, and make them interact properly. First, we change the variable for passing requests from the message-handling actions (line 12-13 in Figure 2b) to a logging action (line 7-8 in Figure 3b). We then model the logging action of the thread and refactor the message-handling actions to make them interact with the logging action.

```
1   FollowerProcessNEWLEADER_UpdateEpoch(i, j) ==
2     /\ IsON(i) /\ IsFollower(i) /\ IsMyLeader(i, j)
3     /\ PendingNEWLEADER(i, j)
4     /\ currentEpoch[i] /= acceptedEpoch[i] /\ ...
5 ① /\ currentEpoch' = [currentEpoch EXCEPT ![i] = acceptedEpoch[i]]
6     /\ UNCHANGED <<history, packetsSync, msgs, ...>>
```

(a) Action 1: Updating the current epoch

```
1   FollowerProcessNEWLEADER_LogAsync(i, j) ==
2     /\ IsON(i) /\ IsFollower(i) /\ IsMyLeader(i, j)
3     /\ PendingNEWLEADER(i, j)
4     /\ currentEpoch[i] = acceptedEpoch[i]
5     /\ packetsSync[i].notCommitted /= << >>
6     /\ LET packetsInSync == packetsSync[i].notCommitted
7        IN /\ queuedRequests' = [queuedRequests EXCEPT
8                                    ![i] = @ \o packetsInSync]
9 ②        /\ packetsSync' = [packetsSync EXCEPT
10                                   ![i].notCommitted = << >>]
11    /\ UNCHANGED <<currentEpoch, msgs,...>>
```

(b) Action 2: Queuing requests for asynchronous logging

```
1   FollowerProcessNEWLEADER_ReplyAck(i, j) ==
2     /\ IsON(i) /\ IsFollower(i) /\ IsMyLeader(i, j)
3     /\ PendingNEWLEADER(i, j)
4     /\ currentEpoch[i] = acceptedEpoch[i]
5     /\ packetsSync[i].notCommitted = << >>
6     /\ LET msg == msgs[j][i][1]
7            m_ackld == [mtype |-> ACK, mzxid |-> msg.mzxid]
8 ③      IN Reply(i, j, m_ackld)
9     /\ UNCHANGED <<currentEpoch, history, ...>>
```

(c) Action 3: Sending ACK to the leader

Figure 3. Fine-grained modeling that splits the atomic action, `FollowerProcessNEWLEADER` in Figure 2b, into three actions (Actions 1–3).

Figure 4a shows the modeling of the asynchronous logging, including an additional variable `queuedRequests` and a new action `FollowerSyncProcessorLogRequest`. The variable `queuedRequests` models the implemented queue (line 2-3 in Figure 4b) that stores the requests to be logged (line 4-6 in Figure 4b). The action `FollowerSyncProcessorLogRequest` (line 2-10 in Figure 4a) focuses on the logic of logging requests (line 9-12 in Figure 4b). It takes out a request from `queuedRequests`, logs it to disk and sends `ACK` to the leader. In this way, we are able to model the actions that are concurrently executed by different threads in a node, for example, `FollowerSyncProcessorLogRequest` (Figure 4a). The interleavings of these locally concurrent actions can then be explored at the model level.

3.2 Coarse-Grained Specifications

When model checking a target module with the fine-grained specification, the other modules are coarsened to reduce state space and avoid state-space explosion. To ensure verification safety, the coarsening must follow the *interaction preserving* principle, i.e., for each module, only the internal part can be omitted while the interactions with other modules must be preserved, such that another module cannot distinguish whether it is interacting with the original or a coarsened

```
VARIABLES queuedRequests
FollowerSyncProcessorLogRequest(i, j) ==
  /\ IsON(i) /\ IsFollower(i) /\ IsMyLeader(i, j)
  /\ queuedRequests[i] /= << >>
  /\ LET toBeSaved == queuedRequests[i][1]
         m_ack     == [mtype |-> ACK, mzxid |-> toBeSaved.zxid]
     IN /\ history' = [history EXCEPT ![i] = Append(@, toBeSaved)]
        /\ queuedRequests' = [queuedRequests EXCEPT ![i] = Tail(@)]
        /\ Send(i, j, m_ack)
  /\ UNCHANGED <<state, acceptedEpoch, ...>>
```

(a) Fine-grained specification in TLA+

```
class SyncRequestProcessor extends ZooKeeperCriticalThread {
  private final BlockingQueue<Request>
    queuedRequests = new LinkedBlockingQueue<>();
  public void processRequest(final Request request) {
    queuedRequests.add(request);
  }
  public void run() {
    while (true) { // Processing every request
      Request si = queuedRequests.poll(…);
      if (si == null) { flush(); si = queuedRequests.take(); }
      if (zks.getZKDatabase().append(si)) {...}
      ... // flush to disk and reply ACK if needed
    } ...
} // zookeeper-server/src/.../server/SyncRequestProcessor.java
```

(b) Code implementation in Java

Figure 4. Fine-grained modeling on concurrency with async logging. `queuedRequests` is used in Figure 3b.

module. This indistinguishability ensures the correctness of compositional model checking.

Safe coarsening is done by following the rationale of interaction preservation in TLA+. In TLA+, we define the global states of a distributed system with *variables* and update the states with *actions*. We define *dependency variables* of an action as the variables in the enabling condition of the action; the dependency relation is transitive—if a dependency variable is calculated from another variable, that variable is also a dependency variable. The dependency variables of a module hence consist of dependency variables of all the actions in the module, where a *module* is a set of actions. We define *interaction variables* as dependency variables shared by two modules.[1]

The coarsening preserves interaction if: (1) all dependency variables of the target module, as well as all interaction variables, remain unchanged after the coarsening; (2) all the updates of the dependency variables and interaction variables remain unchanged after the coarsening. These two constraints relate actions of a fine-grained module to those of the coarsened module through dependency variables and interaction variables along with their updates.

[1]Our definition of interaction variables is conservative, because dependency variables in two modules may not convey any interaction. In practice, this case is rare so we make the definition concise and easy to use.

We denote a specification S that consists of n modules as $S = \bigcup_{1 \le i \le n} M_i$, and $\widetilde{M_i}$ as a module obtained by coarsening M_i following the above two constraints. S_i is denoted as the specification by coarsening every other module except M_i, i.e., $S_i = (\bigcup_{j \ne i} \widetilde{M_j}) \cup M_i$.

Let the traces allowed by S and S_i be T_S and T_{S_i} respectively. When we are only concerned with the states of the target module M_i, all traces in T_S and T_{S_i} are projected to M_i, which are denoted as $T_S|_{M_i}$ and $T_{S_i}|_{M_i}$. Then we can talk about the equivalence relation between traces with respect to a target module, which is defined as: $T_S \overset{M_i}{\sim} T_{S_i} \overset{def}{=\!=} T_S|_{M_i} = T_{S_i}|_{M_i}$.

The safety of the coarsening is captured by the equivalence between traces, as in the following theorem.

Interaction Preservation Theorem. *Given* $S = \bigcup_{1 \le i \le n} M_i$ *and* $S_i = (\bigcup_{j \ne i} \widetilde{M_j}) \cup M_i$, *we have* $T_S \overset{M_i}{\sim} T_{S_i}$.

Appendix B provides the proof sketch of the theorem.

The key concept of the theorem is inspired by [47] but is used differently. In [47], interaction preservation is used to establish the abstraction-refinement relation between different levels of specifications. In this work, coarse-grained and fine-grained specifications do not have abstraction-refinement relations due to model-code gaps. Therefore, coarsening does not enforce abstraction relations. Both fine-grained and coarse-grained specifications are checked against implementation for conformance. The correctness of modules under verification is guaranteed by ensuring invariants of system design. (The correctness is not guaranteed by abstraction relations between fine-grained and the coarse-grained specifications as in [47].) The interaction preservation ensures that all possible behaviors of the target module under verification is systematically explored, without noticing that internal details of its interacting modules are omitted.

In our experience, identifying interaction variables in TLA+ specifications is straightforward, especially for systems designed with modularity and loose coupling. We can also potentially borrow ideas from implementation-level model checking [50] to dynamically identify interaction.

Case study: Coarsening the model of the Election and Discovery phases. In ZooKeeper's system specification, the Election and Discovery phases are modeled by eight atomic actions (Figure 5a). An action may handle an incoming message (e.g., `LeaderProcessACKEPOCH`), send a message to a peer (e.g., `ConnectAndFollowerSendFOLLOWERINFO`), or broadcasts messages to peers (e.g., `FLEHandleNotmsg`). If these two phases are not the target of the verification, we can coarsen the eight actions into one action, as shown in Figure 5b.

To do so, we first identify internal variables that do not interact and thus do not affect other phases. For example, one internal variable is `currentVote` (line 7 in Figure 5a). This variable stores the leader information which is only consumed by the local node; hence, this variable can be abstracted away in the coarse-grained action.

```
 1    (* Actions in Election phase *)
 2    FLEReceiveNotmsg(i, j) == ...
 3    FLENotmsgTimeout(i) == ...
 4    FLEHandleNotmsg(i) == ...
 5    FLEWaitNewNotmsg(i) == /\ ...
 6      /\ state' = [state EXCEPT ![i] =
 7 ①                 IF currentVote[i].proposedLeader = i
 8                   THEN LEADING ELSE FOLLOWING]
 9    (* Actions in Discovery phase *)
10    ConnectAndFollowerSendFOLLOWERINFO(i, j) == ...
11    LeaderProcessFOLLOWERINFO(i, j) == ...
12    FollowerProcessLEADERINFO(i, j) == /\ ...
13 ②  /\ zabState' = [zabState EXCEPT ![i] = SYNCHRONIZATION]
14    LeaderProcessACKEPOCH(i, j) == /\ ...
15 ②  /\ zabState' = [zabState EXCEPT ![i] = SYNCHRONIZATION]
```

(a) Before coarsening (eight actions in two modules)

```
1    ElectionAndDiscovery(i, Q) ==
2      /\ i \in Q /\ IsQuorum(Q) /\ ...
3      /\ state' = [s \in Server |-> IF s = i THEN LEADING
4 ①                  ELSE IF s \in (Q\{i}) THEN FOLLOWING ELSE state[s]]
5      /\ zabState' = [s \in Server |-> IF s = i \/ s \in (Q\{i})
6 ②                       THEN SYNCHRONIZATION ELSE zabState[s]]
```

(b) After coarsening (one action)

Figure 5. Interaction-preserving coarsening of the eight actions in the Election and Discovery phases.

In comparison, variables `state` and `zabState` have external effects; hence, they cannot be abstracted away. We use ① to show how the variable `state` is updated before and after the coarsening. Before coarsening, the `state` of a node is updated according to the leader information, which is stored in `currentVote`. After coarsening, all the participating nodes atomically update their `state` as either `LEADING` or `FOLLOWING` (line 3-4 in Figure 5b). This coarsening is interaction preserving, which is checked based on the constraints. We use ② to show how the variable `zabState` (i.e., the node phase) is updated before and after the coarsening. Before the coarsening, `zabState` is updated in different actions (lines 13 and 15 in Figure 5a); after coarsening, all the participating nodes are collectively transitioned to the Synchronization phase (line 5-6 in Figure 5b).

3.3 Composition

Composing modules and their actions is naturally supported by TLA$^+$. The common practice is to define a next-state action that nondeterministically chooses one action from one module to run in each step.

We define four modules corresponding to the four phases of the Zab protocol and compose their specifications into mixed-grained specifications. The four phases are Election, Discovery, Synchronization and Broadcast as shown in Figure 6. Figure 7 shows one example composition, including one (coarsened) action for both Election and Discovery, (fine-grained) actions for Synchronization, and actions for Broadcast. In addition, the next-state action also includes actions for modeling faults (e.g., a node crash). The entire specification is defined as the initial state (`Init`) and the state transition represented by the next-state action (`[]Next`). Note that `[]` is the temporal operator □ which, in this context, means that the next-state action keeps running forever.

This style of composition captures non-deterministic natures of distributed systems. In each step, `Next` chooses any possible action to run since it is defined as the disjunction (`\/`) of all actions. If the action involves certain leader or follower node, `Next` also chooses any possible node using the existential quantifier (`\E`). So, the definition of `Next` allows any (enabled) action with any server to happen at any point.

In this way, we compose different combinations of coarse- and fine-grained modules into different specifications for checking different phases of the Zab implementation.

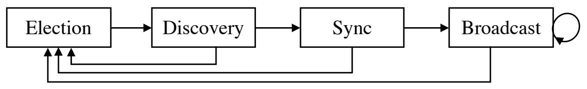

Figure 6. We define four modules and write specifications per module based on phases in Zab.

```
 1    (* Next-state action*)
 2    Next ==
 3      (* Abstracted action for Election and Discovery *)
 4 ①    \/ \E i \in Server, Q \in Quorums: ElectionAndDiscovery(i, Q)
 5 ↓    (* Actions in Synchronization *)
 6 ②    \/ \E i, j \in Server: FollowerProcessNEWLEADER_UpdateEpoch(i, j)
 7      \/ \E i, j \in Server: FollowerProcessNEWLEADER_LogAsync(i, j)
 8      \/ \E i, j \in Server: FollowerProcessNEWLEADER_ReplyAck(i, j)
 9      \/ \E i, j \in Server: FollowerSyncProcessorLogRequest(i, j)
10      \/ ...
11 ↓    (* Actions in Broadcast *)
12 ③    \/ \E i \in Server: LeaderProcessRequest(i) \/ ...
13      (* Other actions, e.g., for modeling faults *)
14      \/ \E i \in Server: NodeCrash(i) \/ ...
15    Spec == Init /\ []Next
```

Figure 7. Composition of coarse- and fine-grained actions. ①: the coarsened action for Election and Discovery (Figure 5b). ②: the fine-grained actions for Synchronization (Figure 3 and Figure 4a). ③: the actions for Broadcast.

3.4 Conformance Checking

We ensure that all the TLA$^+$ specifications we wrote, despite their granularities, match the implementation through conformance checking. For a specification, the conformance checker explores model-level state space to generate traces, replays them in the implementation, and compares model- and code-level execution traces (§3.5.2).

If the conformance checking detects a discrepancy between the model and the implementation, we debug the discrepancy and revise the specification to match code behavior. The deterministic replay provided by our conformance checker makes it easy to debug at the code level. After the specification is updated, we run a new round of conformance checking until it passes. The continuous conformance checking helped us find several discrepancies, ranging from inconsistent message types between specification and implementation, to incorrect conditional branches and wrong variable assignments that lead to unrealistic state transitions.

Limitation. The conformance checking is unsound. It could miss behaviors in implementation that are not modeled by the specification, while the discrepancy is not detected during conformance checking. We consider improving it with refinement checking and abstraction mapping [44, 95]. We only check safety properties instead of liveness properties (which are hard for conformance checking).

3.5 Remix: Tooling Support

We wrote Remix as a framework and tooling support for creating mixed-grained specifications by composing multi-grained specifications of different modules, interfacing the model checker (TLC), and providing conformance checking.

3.5.1 Workflow. We select the specification of each module and compose them into a mixed-grained specification for the entire system. We write the specifications in a composable manner (§3.3) so composing them is straightforward. Currently, the selection is done manually; future work can automate it based on time budget of model checking. If there is no specification at the desired granularity, one can write a new specification (see §3.1 and §3.2). The new specification will then be added into Remix. In case that the specifications are not composable, parsing or semantic errors will be reported by the TLC model checker.

With multi-grained specifications, a specification is associated with protocol- and code-level invariants (§4.2). Remix automatically selects invariants when composing specifications, which will be checked during model checking.

We run continuous conformance checking to ensure the specification in Remix is synchronized with the implementation (§3.4). We describe our conformance checker which is built on top of deterministic execution.

3.5.2 Conformance checker. The conformance checker randomly explores the model-level state space to obtain a set of traces under a predefined time budget (e.g., 30 minutes). For each trace, the conformance checker deterministically replays it at the code level (§3.5.3) and reports a discrepancy if (1) a model-level variable and its code-level counterpart have different values, or (2) a model-level action's code-level counterpart, once enabled, never takes place (in 50 seconds).

Our conformance checker is not guaranteed to detect all discrepancies through random exploration, similar to prior work [35, 40, 94]. To avoid false alarms caused by discrepancies, Remix deterministically replays each model-level trace that violates some safety property to confirm the safety violation also happens in the implementation.

Our conformance checker also reports implementation bugs with obvious symptoms like assertion failures when replaying traces. Developers can mark such traces and later during model checking Remix will focus on exploring other traces since the marked traces are already known as buggy.

3.5.3 Deterministic execution. The conformance checker needs to deterministically replay model-level traces at the code level. Remix realizes deterministic execution by having a central coordinator that intercepts and coordinates actions from different threads on different nodes. The coordinator takes a model-level trace as the input, schedules the code-level actions one by one accordingly, and injects faults (e.g., node crashes) when needed.

To deterministically replay a model-level trace, the coordinator needs to map each model-level action to the code, and precisely control the interleaving between code-level actions. Remix currently requires developers to provide a mapping from each model-level action to the events that represent the beginning and the end of the corresponding code-level action. Remix then instruments around such events to control the interleaving. For example, the model-level action `FollowerProcessNEWLEADER(f,l)`'s corresponding code-level action begins with the leader `l` calling the `writeRecord` method to send a `NEWLEADER` message to the follower `f`, and ends with `f` sending back an `ACK` message (the `writeRecord` method is used for sending messages between ZooKeeper nodes). Developers provide this mapping and then Remix instruments `writeRecord` to inject an RPC client that calls the coordinator during runtime with the context information (e.g., arguments and the caller of `writeRecord`). The call returns only when the coordinator schedules this action according to the model-level trace. The coordinator will not schedule any other actions until the currently running action ends. In this way, the coordinator deterministically decides when each code-level action begins and controls the interleaving between code-level actions.

If the implementation can interleave events that appear atomic at the model level and generate discrepant states, developers must either revise the specification to enable the code-level interleaving, or provide a more precise mapping to make the conformance. For example, the election messages can interleave during Election, with a non-deterministic leader generated. For a coarsened Election action that elects a target leader at the model level, developers can set the messages that vote for the target leader with higher priority. In this way, the deterministic replay is able to generate a matched state required by the model-level action. The debugging process is iterative until the specification and the mapping reach satisfactory conformance.

The deterministic execution is also useful for debugging safety violations. For a trace that violates a safety property during model checking, Remix deterministically reproduces it at the code level, so that developers can diagnose the root cause of the safety violation.

Remix implements the deterministic execution coordinator using the Java Remote Method Invocation [23] framework and instruments ZooKeeper using AspectJ [18] to inject the RPC clients. For each version of the specification, developers need to provide a mapping from each model-level action to the corresponding code-level action.

An alternative option for realistic execution is to intercept system calls [94]. Intercepting system calls avoids system-specific instrumentation and leaves the target system unmodified. However, we choose to instrument the target system because it is hard to control user-level threads (e.g., event handlers in ZooKeeper) accurately at the system call level.

4 Verifying ZooKeeper

We wrote the protocol specification of Zab (§2.1.1), as well as the system specification of ZooKeeper (§2.1.2), including the specification of the Fast Leader Election (FLE) which was not a part of the original Zab protocol [61]. We use the TLC model checker to verify both of them. We build on the system specification as the baseline to develop code specifications and compose mixed-grained specifications as discussed in §3. The mixed-grained specifications are then used to verify the implementation of ZooKeeper and its code changes.

4.1 Mixed-Grained Specifications for Log Replication

We present the mixed-grained specifications composed using Remix to verify the log replication of ZooKeeper. We focus on log replication because it is the main procedure for achieving consensus in ZooKeeper.[2] It involves both the Synchronization and Broadcast modules. The log replication implementation has been greatly optimized during the evolution of ZooKeeper; hence, the code, especially the Synchronization module, can be described by neither the protocol nor the system specifications effectively. As a result, severe bugs (e.g., those that lead to data loss or inconsistencies) were constantly reported, such as [1, 3–5, 7].

Our first step is to ensure the system specifications match the ZooKeeper implementation. We run conformance checking and find a discrepancy where the model-level traces cannot be successfully replayed at the code level due to an unexpected exception. The root cause is that at the code level, if the follower receives the `COMMIT` message *after* the `NEWLEADER` message, it will throw an unexpected `NullPointerException` and terminate the Synchronization phase. The issue had been reported in ZK-4394 [8] but was (still is) not resolved. So, we adjusted the specification by adding new conditions and a new commit assertion to check the `NullPointerException` and avoiding further exploration once ZK-4394 occurs.

Based on the conformed system specification, we create the following mixed-grained specifications (Table 1).

- **mSpec-1.** Since we target log replication, we coarsen the Election and Discovery modules. Section 3.2 described the coarsening which coarsens the eight actions of the Election and Discovery modules in the system specification into one `ElectionAndDiscovery` action.
- **mSpec-2.** We write a fine-grained specification of the Synchronization module to model the non-atomic updates of epoch and history when a follower receives the `NEWLEADER` message (§3.1). So, the model checker can explore intermediate states between the updates of epoch and history, which is induced by node crashes. As we focus on log replication, mSpec-2 uses the coarsened action for the Election and Discovery modules in mSpec-1.
- **mSpec-3.** This specification further models multithreading concurrency in log replication. There are three main threads in the follower process, for handling incoming messages, logging transactions, and committing transactions, respectively. We distinguish the asynchronous actions executed by different threads, and specify them into separate actions, as demonstrated in §3.1. mSpec-3 also uses the coarsened `ElectionAndDiscovery`.
- **mSpec-4.** As a reference, we create mSpec-4 by composing system specifications of the Election and Discovery modules and fine-grained log replication modules in mSpec-3.

[2]We also verify the other part of ZooKeeper and find bugs including ZK-2776, ZK-3336, ZK-3707, ZK-4040, ZK-4416 and ZK-4781. These bugs are known bugs but still exist in the checked versions.

Spec	Election	Discovery	Log Replication	
			Synchronization	Broadcast
SysSpec	Baseline	Baseline	Baseline	Baseline
mSpec-1	Coarsened		Baseline	Baseline
mSpec-2	Coarsened		Fine-grained (atom.)	Baseline
mSpec-3	Coarsened		Fine-grained (atom.+ concur.)	Fine-grained (concur.)
mSpec-4	Baseline	Baseline	Fine-grained (atom.+ concur.)	Fine-grained (concur.)

Table 1. Mixed-grained specifications for verifying log replication, composed from multi-grained specifications. "SysSpec" refers to the system specification that passes conformance checking (used as the baseline).

4.2 Invariants

We specify 14 invariants that are checked throughout the model checking, as shown in Table 2. Ten invariants are safety properties defined by the Zab protocol [60], including both core properties and lemmata; these invariants must be satisfied by any Zab implementations. During model checking, these ten invariants are checked upon state transition. These invariants apply to specifications of any granularity.

We also define other four types of invariants (11 instances in total) based on the code-level implementation of ZooKeeper. We observe that developers add additional checks on specific behavior (e.g., by throwing exceptions and using assertions). Some of them are not reflected by the safety properties of the Zab protocol, and thus shall be included in fine-grained specifications. Some of them are caused by known, but still not resolved bugs in the code, such as ZK-4394 discussed in §4.1. Essentially, each of these invariants specify the execution is

ID	Invariant(s)	Source
I-1	**Primary uniqueness.** There is at most one established leader for each epoch.	Protocol
I-2	**Integrity.** If some process delivers t, then some primary has broadcast t.	Protocol
I-3	**Agreement.** If some process f delivers t, and some process f' delivers t', then f' delivers t or f delivers t'.	Protocol
I-4	**Total order.** If some process delivers t before t', then any process that delivers t' must also deliver t and deliver t before t'.	Protocol
I-5	**Local primary order.** If a primary broadcasts t before it broadcasts t', then a process f that delivers t' must also deliver t before t'.	Protocol
I-6	**Global primary order.** If a process f delivers both t (in epoch e) and t' (in epoch e', $e < e'$), then f must deliver t before t'.	Protocol
I-7	**Primary integrity.** If a primary ρ_e broadcasts t and some process f delivers t' st. t' has been broadcast by $\rho_{e'}$, $e' < e$, then ρ_e must deliver t' before it broadcasts t.	Protocol
I-8	**Initial history integrity.** Let e, e' be epochs, $e < e'$, and e be an established epoch. $I_e \sqsubseteq I_{e'}$.	Protocol
I-9	**Commit consistency.** Let Δ_f be the delivered transaction sequence of process f, and $f.e$ be f's last committed epoch. $I_{f.e} \sqsubseteq \Delta_f$.	Protocol
I-10	**History consistency.** For any two processes f and f' that participate in epoch e, either $h_f \sqsubseteq h_{f'}$ or $h_{f'} \sqsubseteq h_f$.	Protocol
I-11	**Bad states (4 instances).** Exceptions or false assertions on the server states upon receiving certain types of messages.	Code
I-12	**Bad acknowledgments (2 instances).** Exceptions or false assertions on the `ACK` message content processed by the leader.	Code
I-13	**Bad proposals (2 instances).** Exceptions or false assertions on the `PROPOSAL` message content processed by the follower.	Code
I-14	**Bad commits (3 instances).** Exceptions or false assertions upon handling the `COMMIT` message or committing a transaction.	Code

Table 2. Invariants including safety properties of the Zab protocol and the code-level assertions by developers. t: a transaction; h_f: (transaction) history of process f; ρ_e: primary of epoch e; $\sqsubseteq$: the relation of prefix; I_e: initial history of epoch e.

Specification diff.	Lines	Variables	Actions	Instr.	Hour
mSpec-1 − SysSpec	+64, -342	29 (-8)	16 (-7)	31 (+0)	18
mSpec-2 − mSpec-1	+34, -19	29 (+0)	17 (+1)	32 (+1)	8
mSpec-3 − mSpec-2	+188, -118	31 (+2)	19 (+2)	36 (+4)	40

Table 3. Efforts of writing multi-grained specifications. "#Instr." refers to the number of instrumentation pointcuts.

on an error path. These invariants are checked whenever the model checker reaches the corresponding execution path. Note that code-level invariants are specific to certain granularities that model the corresponding execution.

4.3 Efforts

Table 3 shows the efforts of writing multi-grained specifications in Table 1. The efforts of writing and maintaining multi-grained specifications are manageable, especially when baseline specifications are available. Fine-grained modeling and coarsening can be done incrementally on top of the reference specification. For example, the differences between the specifications are less than 500 lines. Following composable formal methods, all invariants and most variables in the baseline specification are directly reused. In addition, we need to provide a mapping from the newly added (coarse- or fine-grained) model-level actions to the code-level actions so that Remix can instrument code for deterministic execution (§3.5.3) at different granularities. Overall, the effort is done within 40 person-hours, and is done by one person who is familiar with the ZooKeeper code and is proficient in TLA$^+$. Further, as more specifications are written, the reusability of composable components grows higher, amortizing the cost.

4.4 Setup

We use a configuration of a three-node ZooKeeper cluster with up to four transactions, up to three node crashes, and up to three network partitions. This configuration is a common practice used in prior work [69, 76, 94, 97]. We use TLC to run model checking on TLA$^+$ specifications on a single machine. We use TLC's breath-first search (BFS) as the strategy for state-space exploration. With BFS, once an invariant is violated, we can obtain the buggy trace with minimal depth.

5 Results and Experience

5.1 Verification Results

We start to systematically model check ZooKeeper using mixed-grained specifications since version v3.9.1. ZooKeeper did not pass the verification. The model checking exposes a total of six severe bugs, as shown in Table 4. All these bugs have serious consequences, including data loss, data inconsistencies, and data synchronization failures. All these bugs are deep bugs, as their manifestations take minimal depths of tens of actions and more than tens of thousands of states, which are hard for developers to reason about. For the same reason, they are also hard to fix (§5.3).

Those bugs are found when TLC reports violations of invariants in the traces during model checking. We then confirmed the bugs by deterministically replaying the traces at the code level using Remix (§3.5). To debug a violation, we analyze the model-level trace to understand the triggers and locate the root cause in the code. The debugging is eased by traces with minimal depth explored by the BFS strategy. In practice, we start the model checking from a small configuration (e.g., one node crash) to a large one (up to three node crashes), which helps us obtain a simple and concise trace.

Table 4 shows the most efficient specification that found each bug. All the bugs except one (ZK-4394) require mSpec-2 and mSpec-3. In other words, finding these bugs needs fine-grained modeling of non-atomic actions and local concurrency; these bugs cannot be found by the baseline system specification (§2.1.2). The results show the importance of closing the model-code gap with fine-grained specifications

Bug ID	Impact	Spec.	Time	Depth	#States	Inv.
ZK-3023	Data sync failure	mSpec-3	11 sec	13	78,892	I-11
ZK-4394	Data sync failure	mSpec-1*	9 sec	20	14,264	I-14
ZK-4643	Data loss	mSpec-2	17 sec	21	208,018	I-8
ZK-4646	Data loss	mSpec-3	109 sec	21	2,880,498	I-8
ZK-4685	Data sync failure	mSpec-3	10 sec	12	67,418	I-12
ZK-4712	Data inconsistency	mSpec-3	11 sec	13	73,293	I-10

Table 4. Bug detection in ZooKeeper v3.9.1. "Spec." shows the most efficient mixed-grained specification to find the bug. "Inv." shows the first violated invariant triggered by the bug. ZK-4394 is not fixed yet so we masked it in our specifications; mSpec-1* refers to the specification before masking it.

that reflect code-level behavior. We discussed ZK-4394 in §4.1. It can be found by mSpec-1 with conformance checking. The bug can be found with the system specification also; but, compared with the system specification, mSpec-1 significantly reduces the time to find the bug. In fact, all the bugs were found in less than two minutes with the benefits of specification coarsening. We discuss more about the efficiency of coarse-grained specifications in §5.2.

Three of the six bugs violate protocol-level invariants and the others violate code-level invariants. We find that code-level invariants are important, as developers often directly throw exceptions to abort the execution in certain critical cases, which is not defined in the protocol (§4.2).

Appendix A describes these bugs in more details.

5.2 Efficiency

Mixed-grained specifications effectively improve efficiency of verifying the target modules. We evaluate the verification efficiency of the five specifications in Table 1. We set the time budget to be 24 hours and the violation limit to be 10,000. We use each of these specifications to verify ZooKeeper v3.7.0. We run TLC (v1.7.0) with the BFS mode for exploration. All the experiments were run on an Ubuntu 22.04 server with two AMD EPYC 7642 processors at 3.3GHz; each processor has 48 cores and 96 hyperthreads. Each specification is checked by 16 workers (threads) with 32 GB memory.

Table 5 shows verification efficiency results in two modes: (5a) stopping at the first violation, and (5b) running to completion (till the limit). The mixed-grained specifications with fine-grained modeling (mSpec-2, -3, and -4) detect violations within the time limit. The baseline (system specification) and mSpec-1 find no violation because of not modeling fine-grained behavior. The baseline and mSpec-4 cannot finish in 24 hours, and we observe that TLC spends most of the time in the Election module without reaching other modules (the leader election algorithm is complex and takes many steps). mSpec-4 costs 2793× more time to detect the first violation compared to mSpec-3 as it does not coarsen the Election and Discovery modules. In comparison, mSpec-1, -2, and -3 coarsen the Election and Discovery modules, enabling TLC to more efficiently check the log replication modules. As a result, these three mixed-grained specifications can finish in tens of minutes, with mSpec-2 and -3 finding the first violation in minutes. So, mixed-grained specifications provide the flexibility to help the model checker focus on target modules with fine-grained models, which is critical to finding bugs and receiving prompt feedback.

Spec	Time	Depth	# States	# Violated Inv.
Baseline	>24h	26	2,271,335,268	None
mSpec-1	12m20s	56	17,586,953	None
mSpec-2	1m15s	21	2,237,960	I-8
mSpec-3	11s	13	77,179	I-10
mSpec-4	8h32m6s	24	967,810,552	I-10

(a) Stopping at the first violation

Spec	Time	Depth	# States	# Violation	# Vio. Inv.
Baseline	>24h	26	2,271,335,268	0	None
mSpec-1	12m20s	56	17,586,953	0	None
mSpec-2	15m55s	62	24,211,064	1,404	I-8
mSpec-3	5m10s	21	1,727,234	>10,000	I-10, I-11, I-12
mSpec-4	>24h	26	2,478,453,900	35	I-10, I-11, I-12

(b) Running to completion

Table 5. Verification efficiency of specifications with different granularities. The configuration is three servers, two transactions, two crashes, and two partitions. "Depth" refers to the number of state transitions; "States" refers to the *distinct* states explored (and reported) by TLC.

$$\begin{tikzcd}
 & \text{ZK-4643} & \text{ZK-4394} \\
 & \text{ZK-3911*} \arrow[ur] \arrow[r] \arrow[dr] & \text{ZK-4646} \\
\text{ZK-2678* (Optimizations of data recovery)} \arrow[uur] \arrow[ur] \arrow[r] \arrow[dr] \arrow[ddr] & \text{ZK-3023} & \text{ZK-4685} \\
 & \text{ZK-3642*} \arrow[r] & \text{ZK-4712} \\
 & \text{ZK-2845*} &
\end{tikzcd}$$

Figure 8. Bugs introduced in ZooKeeper's log replication implementation. * refers to those with fixes merged.

5.3 Verifying Bug Fixes

Fixing bugs is challenging—it is easier to prevent specific symptoms, but harder to rule out root causes due to the complexity of reasoning about all interleavings of actions. We first fixed ZK-4712, but found the other bugs are much harder to fix. They are rooted in various performance optimizations since 2017, triggered by ZK-2678 [2]. These optimizations have introduced over ten data loss/inconsistency bugs [3, 4, 6–8, 11–13, 15, 16, 26]. Some of them were fixed, while others were not. Without sufficient verification at the time, some accepted fixes introduced new bugs (Figure 8).

Mixed-grained specifications enable us to verify bug fixes efficiently. We verify four Pull Requests (PRs) that attempted to fix the bugs in Table 4. All the PRs use multithreading with non-atomic updates of epoch and history in the Synchronization phase. Therefore, we use the specification of mSpec-3 as the base specification. With the fix of ZK-4712, we updated mSpec-3, referred to as mSpec-3^+ (verified using TLC). For

	Spec.	Change	Time	Depth	#States	Inv.
PR-1848	mSpec-3$^+$	+68, -29	274s	21	8,166,775	I-8
PR-1930	mSpec-3$^+$	+102, -66	17s	13	270,881	I-12
PR-1993	mSpec-3$^+$	+71, -51	34s	15	765,437	I-11
PR-2111	mSpec-3$^+$	+70, -43	38s	15	808,697	I-11

Table 6. Verifying bug fixes (pull requests). mSpec-3$^+$ is the specification of mSpec-3 with the fix of ZK-4712.

each PR, we map the code changes to mSpec-3$^+$ and update mSpec-3$^+$ accordingly. Table 6 shows the verification results with the mode of stopping at the first violation. All four fixes are detected with invariant violations within five minutes.

The challenge of fixing these bugs is that all of them involve the logic of handling the NEWLEADER message. So, it is hard to fix them in isolation—the fix of one bug can make it hard to fix the others. For example, PR-1993 targets ZK-4646 and ZK-4685, without considering other bugs like ZK-4394. To further fix ZK-4394 based on PR-1993 need heavy revision on the code logic of PR-1993. If we consider the full picture, PR-1993 is not a good step towards a complete solution. Some fixes lead to new bugs, e.g., the merged fix for ZK-3911 did not prevent the same violation, but opened new triggering paths of ZK-3023 and induced new bugs like ZK-4685.

The mixed-grained specifications help us understand the root causes and systematically verify whether a given code fix is completed by exhaustively exercising all the possible interleavings modeled by the fine-grained specification. With a holistic understanding, we developed a fix that resolves *all* the bugs in Figure 8, and verify it with extensive model checking. We discuss our resolution in §5.4. The verified fixes have been merged to the latest version of ZooKeeper.

5.4 Improving The Zab Protocol

One essential reason for the error-proneness of log replication is rooted in deviation of the implementation in ZooKeeper from the Zab protocol. For example, the atomicity of updating epoch and history is explicitly required by the protocol, but is not followed by the implementations.[3] Basically, the protocol no longer guides the implementation. To fix existing bugs and also make it easy to implement correctly, we remove the atomicity requirement of the two updates from the Zab protocol but require their order—the follower updates its history *before* updating its epoch.

We update the protocol specification (§2.1.1), which splits the action of handling a NEWLEADER message into two serialized actions of updating history and epoch. In this way, the model checker can explore traces when a follower crashes right after it updates history. We add a variable `servingState` to help express the enabling conditions of updating history and epoch. We run extensive model checking with TLC to verify the new protocol specification and it passes all the ten protocol-level invariants (Table 2).

[3]The Zab protocol implemented in ZooKeeper [86] has evolved in several other ways different from original papers [60, 61].

We then update the implementation based on the new protocol, and address all the bugs in Figure 8 by enforcing the order of several critical events. We change asynchronous logging into synchronous logging and the overhead is acceptable as it only occurs when the follower receives COMMIT before handling the NEWLEADER message (synchronous logging is already used in v3.9.2). The new implementation conforms to the new protocol specification. We updated mSpecs in Table 1 which passed model checking.

6 Discussion

Writing TLA$^+$ specifications for distributed protocols and systems has become a common practice [31, 36, 52, 65, 74, 81, 106]. As articulated in [81], writing specifications enforces clear thinking, precise designs, and unambiguous documents. We started from the protocol and system specifications (§2.1). However, it is clear to us that code-level implementation evolves fast and inevitably deviates from the protocol and system designs due to performance optimizations. As a consequence, deep bugs often reside in the model-code gaps (§2.2). We advocate for reconciling formal specifications with code implementations by fine-grained modeling of several important implementation aspects, such as multithreading and non-atomic updates. We show that such modeling is beneficial and helps understand and address a few complex, long-lasting issues in ZooKeeper.

We cope with the enlarged state space introduced by fine-grained modeling with mixed-grained specifications which are composed of multi-grained specifications. In essence, mixed-grained model checking is a divide-and-conquer strategy which leverages good modularity and loose coupling of distributed systems like ZooKeeper. Apart from model checking, multi-grained specifications also bring benefits such as enforcing precise thinking, communication, and documentation across the ladder of abstractions (§5.3).

Compared with implementation-level model checking [50, 69, 76, 101], our philosophy is different. We intend to do checking at the model level, which is more efficient and scalable than running heavy distributed system code. It is harder to reduce unnecessary overheads at the code level such as network and disk operations and code that is not targeted for verification. Prior work [50] developed dynamic interface reduction, but can only reduce to *node* locally. Our goal is to develop models that effectively reflect the code and have the flexibility to choose granularities by *phases*. In our experience, it is not easy to define a clean local/non-local boundary for nodes in existing TLA$^+$ specifications due to the common use of global variables in TLA$^+$.

Amazon recently shared their practice of using property-based testing with randomly generated test input to exercise both the implementation and the executable model and checks whether they agree [35]. We cannot directly apply this approach: (1) our model in TLA$^+$ is not executable and

(2) existing property-based testing framework [22] cannot directly control event interleaving *inside* the system under test. So, our conformance checker follows a top-down approach, i.e., using the TLC model checker to generate event traces and replays them in the implementation by deterministically controlling event interleaving, as used in SandTable [94]. Conformance can also be checked with a bottom-up approach, i.e., generating implementation-level traces and checking whether they are allowed by the model, as adopted by VYRD [44], CCF [40] and etcd [14]. We choose the top-down approach as it can be easily reused for deterministic replay and bug confirmation in the implementation once some safety violation is found at the model level.

We believe that our practice of writing multi-grained specifications is viable and can be generalized beyond ZooKeeper. As we show in §4.3, the efforts on writing multi-grained specifications and instrumentations are reasonable and can be amortized, and the specifications are written by one person who knows deeply about the protocol and the implementation. On the other hand, maintaining multi-grained models can be more expensive. We have no silver bullet beyond running continuous conformance checking upon code changes and updating the specifications accordingly. Modern continuous integration is an opportunity to update models incrementally. Since most code changes are local, most modules and high-level specifications stay unchanged. The cost of running conformance checking upon changes and updating specifications accordingly mainly lies in the changed modules and fine-grained specifications. The mix-grained models can make continuous verification more efficient.

7 Related Work

We documented our preliminary work in [82]. At that time, we were writing three types of specifications: (1) protocol specification, (2) system specification as super-doc, and (3) test specification for testing of ZooKeeper. Those specifications were not coherent and could not meet the goal of verifying ZooKeeper implementation in this paper.

In [82], we defined a test specification as a refinement of a system specification. However, we later find that refinement is not the right approach to addressing model-code gaps (§2.2); thus we no longer enforce refinement relations. Instead, we write fine-grained specifications to close model-code gaps and write coarse-grained specifications to speed-up exploration of global states. For the same reason, unlike in [82] where we viewed a system specification as an abstraction of a test specification, we no longer enforce the abstraction relation during coarsening but only ensuring interaction preserving. For example, when specifying atomicity-related behavior, the coarse-grained specification does not abstract away any variable from the fine-grained one, but has fewer transitions compared to the fine-grained one, so no obvious abstraction or refinement relation exists.

Verification. TLA$^+$ has been widely used for modeling and verifying distributed systems. Recent work [52, 81] uses TLA$^+$ to model and verify system designs, but not implementations. Recently, TLA$^+$ based techniques have also been developed to test and verify distributed system implementations [82, 94, 97]. From the tooling perspective, SandTable [94] is a close approach. Like Remix, SandTable models distributed systems in TLA$^+$, verifies systems using model checking and ensures the specification quality using conformance checking. Therefore, we believe that SandTable can effectively benefit from multi-grained specifications. Notably, we notice that unlike Remix, SandTable cannot check thread interleaving because it intercepts at the system-call level and thus is hard to differentiate between user-level threads. This is one reason we choose to instrument application code so as to control user-level thread interleaving.

Programming languages with built-in model checking support [43, 48, 51, 62, 63, 100] can help build clean-slate verified distributed systems. Unfortunately, it is hard for us to use them for ZooKeeper, which requires major revisions. We were mostly looking for "lightweight" formal methods [35].

Compared to implementation-level model checking [50, 69, 76, 90, 101], we take a different approach by exploring state space at the *model* level to avoid code-level overhead, as discussed in §6. Among implementation-level model checkers, DeMeter [50] also takes a divide-and-conquer strategy and decomposes the problem of model checking a distributed system into model checking each *node* locally. Differently, our approach decomposes the model checking problem into model checking each *phase* and thus is complementary.

Prior work also explored model checking support for distributed systems [54, 68, 87]. In particular, DBSS [87] decomposes model checking by focusing on variables relevant to the property being checked, and verifies protocol-level correctness. Our decomposition is agnostic of any specific property, and we focus on verifying whether the system correctly implements the protocol.

Refinement checking and trace validation are also studied [14, 40, 44, 95]. Tasiran et al. [95] validates hardware designs by connecting the specification and simulation and using model checking to monitor correctness and coverage. VYRD [44] implements I/O and view refinement checking to detect runtime refinement violations for concurrent programs. Our approach verifies the implementation by exploring specification-level states, with the checking of specification-implementation conformance. The conformance checking shares a similar iterative process of selecting commit points in implementation and debugging the mapping between specification and implementation in [44, 95].

Besides, deductive verification approaches have been used to build verified distributed system implementations [55, 57, 88, 89, 92, 98]. Deductive verification does not need a checker to explore state space. However, deductive verification requires hard efforts to write proofs and cannot be directly applied to existing distributed systems.

Bug finding. Many testing techniques [19, 33, 34, 37, 38, 45, 46, 49, 64, 75, 91, 96, 99] detect bugs in distributed systems by fault injection. These tools typically inject faults randomly [83, 96] or focus on a system's vulnerable points with manual guidance [19, 38, 49, 64] or automated analysis [33, 37, 75, 99]. Besides, several projects [72, 73, 105] detect distributed concurrency bugs caused by unexpected interleaving among node events by analyzing happen-before relationships between events [72, 73] or manipulating event ordering [105]. Recent works have also used model checkers to generate test cases for distributed systems [42, 97]. Developers at Amazon have applied property-based testing to test a production system against executable specifications [35]. Despite their effectiveness in detecting bugs, none of these techniques can verify a distributed system by exhaustively exploring its state space.

8 Concluding Remarks

In this paper, we show that formal methods like TLA$^+$ can not only verify protocol and system designs, but also help verify system implementation by modeling important code-level behavior with conformance checking. We advocate for the practice of multi-grained specification and show that the composed mixed-grained specifications provide useful capabilities to manage the state space, making model checking and verification efficient and more usable. With formal methods like TLA$^+$ being widely accepted and adopted, we hope that our work leads to a forward step in empowering formal methods to benefit distributed systems in practice.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Serdar Tasiran, for their insightful comments. Huang's group is supported by the National Natural Science Foundation of China (62025202, 62372222), the CCF-Huawei Populus Grove Fund (CCF-HuaweiFM202304), the Cooperation Fund of Huawei-NJU Next Generation Programming Innovation Lab (YBN2019105178SW38), and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (KYCX24_0235). Xu's group is supported in part by NSF CNS-2130560, CNS-2145295, and a VMware Research Gift.

A Descriptions of Detected Bugs

We provide more information of the bugs in Table 4. The detection of these bugs is described in §5.1. Two of these bugs are known bugs, while the others are new bugs detected during the process of verifying ZooKeeper using Remix. All the bugs are deep safety bugs that are hard to trigger without model checking—each of them takes tens of actions and tens of thousands of states to manifest (see Table 4).

ZK-3023 [4]. The follower fails to catch up with the up-to-date committed data after data recovery is finished. It was caused by the asynchronous commit of the transactions during the Synchronization phase. The bug is triggered when the leader handles `ACK` of `UPTODATE` before the follower commits the pending requests. The bug was known but Remix still detected it in the latest version by then. This bug was originally reported by a test, but the test cannot reliably trigger the buggy interleaving. Our tool deterministically reproduced this bug by detecting violations of I-11 (bad states).

ZK-4394 [8]. This bug could unexpectedly terminate data recovery, which can occur repeatedly and make the follower unavailable. When a follower cannot match the `COMMIT` message to a received request in the Synchronization phase, it throws `NullPointerException`. The bug is triggered when the follower, after processing the `NEWLEADER` message, receives a `COMMIT` message before the `UPTODATE` message. It was a known bug but still in the latest version of ZooKeeper we verified.

ZK-4643 [12]. This bug results in data loss. The implementation fails to guarantee that the follower atomically updates the history with the epoch (an atomic action in the original Zab protocol). It manifests when a follower crashes after updating its epoch, becomes the new leader with stale committed history, and then truncates the committed data of others. The triggering involves a follower crash between the update of epoch and the update of history, together with two crashes of other nodes across three rounds of Election, Discovery, and Synchronization. This is a new bug we detected.

ZK-4646 [13]. This bug causes data loss. The root cause lies in the asynchronous logging of the followers when the leader starts serving clients, which is assumed to be synchronously done by the protocol. It manifests when the uncommitted data is seen by clients and then be removed later. The bug is triggered when leader and followers all crash after a client reads the data that a majority of followers have not persisted in disk, and then one of the follower is elected as the new leader. This is a new bug we detected.

ZK-4685 [15]. This bug fails data recovery among nodes, which increases recovery time and reduces system availability. The root cause is that the leader fails to recognize an `ACK`, which blocks the leader and then leads to the shutdown of all nodes from the Synchronization phase. To trigger this, the follower replies to the leader with `ACK` of `PROPOSAL` before `ACK` of `NEWLEADER` when the leader is collecting a quorum of `ACK`s of `NEWLEADER`. This is a new bug we detected.

ZK-4712 [16]. This bug causes data inconsistency, as a follower keeps extra transactions in its log even after data recovery, making clients obtain inconsistent views from different servers. It was caused by the asynchronous logging during the follower's shutdown. To trigger it, a follower goes back to the Election phase with a non-empty request queue for logging, and then updates its latest transaction ID before processing requests in the queue. It is a new bug we detected.

B Proof Sketch of the Interaction Preservation Theorem

We prove that interaction-preserving coarsening does not affect the correctness of model checking.

We denote a specification S that consists of n modules as $S = \bigcup_{1 \le i \le n} M_i$, and we define $\widetilde{M_i}$ as a module obtained by coarsening M_i following the constraints of interaction preservation (§B.2). S_i is denoted as the specification by coarsening every other module except M_i, i.e., $S_i = (\bigcup_{j \ne i} \widetilde{M_j}) \cup M_i$.

Let the traces allowed by S and S_i be T_S and T_{S_i} respectively. When we are only concerned with the states of the target module M_i, all traces in T_S and T_{S_i} are projected to M_i, which are denoted as $T_S|_{M_i}$ and $T_{S_i}|_{M_i}$. Then we can talk about the equivalence relation between traces with respect to a target module, which is defined as: $T_S \overset{M_i}{\sim} T_{S_i} \overset{def}{=\!=} T_S|_{M_i} = T_{S_i}|_{M_i}$.

The safety of the coarsening is captured by the equivalence between traces, as in the following theorem:

Interaction Preservation Theorem. *Given $S = \bigcup_{1 \le i \le n} M_i$ and $S_i = (\bigcup_{j \ne i} \widetilde{M_j}) \cup M_i$, we have $T_S \overset{M_i}{\sim} T_{S_i}$*

Proof. The basic idea of the proof is that, if the target module M_i cannot distinguish whether it is interacting with the original module M_j or the coarsened module $\widetilde{M_j}$, then the behavior of M_i is not affected by the coarsening.

We define the notations used in the theorem and the proof and present the rule for ensuring interaction preservation. In the proof, we first present the condensation of traces ($T_S|_{M_i}$), to restrict our attention to the target module M_i. Then we establish the equivalence between $T_S|_{M_i}$ and $T_{S_i}|_{M_i}$.

B.1 Notations

TLA$^+$ basics. In the TLA$^+$ specification language, a system is specified as a state machine by describing the possible initial states and the allowed state transitions called $Next$. Specifically, the specification of system design contains a set of *system variables* $\mathcal{V}$. A *state* is an assignment to the system variables. $Next$ is the disjunction of a set of actions $a_1 \lor a_2 \lor \cdots \lor a_p$, where an *action* is a conjunction of several clauses $c_1 \land c_2 \land \cdots \land c_q$. A *clause* is either an *enabling condition*, or a *next-state update*. An enabling condition is a state predicate which describes the constraints the current state must satisfy, while the next-state update describes how variables can change in a step (i.e., successive states).

Whenever every enabling condition ϕ_a of an action a is satisfied in a given "current" state, the system can transfer to the "next" state by executing a, assigning to each variable the value specified by a. We use "$s_1 \xrightarrow{a} s_2$" to denote that the system state goes from s_1 to s_2 by executing action a, and a can be omitted if it is obvious from the context. Such execution keeps going and the sequence of system states forms a trace of system behavior.

A system usually consists of several modules, each implementing some specific function. For the TLA$^+$ specification of a distributed system, we define:

Definition 1 (module)**.** A module is a set of actions. All the modules form a partition of all actions in the specification.

Assume that we write a specification $S = \bigcup_{1 \le i \le n} M_i$ for the system under verification. Our target module is M_i, and we coarsen every other module, obtaining $S_i = (\bigcup_{j \ne i} \widetilde{M_j}) \cup M_i$. The coarsening ensures interaction preservation (§B.2).

We define the set of all possible traces allowed by S (resp. S_i) as T_S (resp. T_{S_i}). T_S and T_{S_i} are different, and T_S usually has a much larger size than T_{S_i}. With target module S_i in mind, we omit unrelated details in the traces by *condensation* (§B.3), and then show that the two sets of traces are equivalent with respect to the target module M_i (§B.4).

B.2 Interaction Preservation

Modules interact with each other through the system variables. To capture this, we first define the *dependency variable* of an action and that of a module:

Definition 2 (dependency variable)**.** Suppose module $M = \{a_1, a_2, \cdots, a_m\}$, dependency variables of M, denoted as $\mathcal{D}_M$, is obtained recursively according to the following rules:

1. For any action $a_i \in M$, its dependency variables $\mathcal{D}_{a_i}$ are the variables which appear in some enabling condition ϕ_{a_i} of a_i.
2. $\bigcup_{1 \le i \le m} \mathcal{D}_{a_i} \subseteq \mathcal{D}_M$. That is, the dependency variables of each action in M belong to $\mathcal{D}_M$.
3. For any $v \in \mathcal{D}_M$ and any action $a_i \in M$, if the next-state update of a_i assigns to v a value calculated from multiple variables (denoted by variable set V_{dep}), then $V_{dep} \subseteq \mathcal{D}_M$. This is due to transitivity of the dependency relation, i.e., if M depends on some variable v and v depends on another variable w, then M also depends on w.

Given the definitions above, we can now say that module M_j interacts with M_i by modifying $\mathcal{D}_{M_i}$.

The notion of dependency variable alone is not sufficient to capture interactions among modules, since even if D_{M_i} are not modified by some action in M_j, M_i may still be affected indirectly. Suppose $x \in \mathcal{D}_{M_i}$, an action in another module M_j assigns to x the value of y (note that y will not be added to $\mathcal{D}_{M_i}$ by the Rule 3 in Definition 2, since x is assigned the value of y in module M_j, not in M_i). In this case, any assignment to y may also change the value of x in subsequent actions. To capture such indirect interactions among modules, we define the set of interaction variables $\mathcal{I}$:

Definition 3 (interaction variable)**.** Suppose the specification contains k modules: $M_1, \cdots, M_k$. The set of interaction variables $\mathcal{I}$ is calculated recursively according to the following rules:

1. $\bigcup_{1\le i<j\le k}(\mathcal{D}_{M_i} \cap \mathcal{D}_{M_j}) \subseteq \mathcal{I}$. That is, if a variable is a dependency variable of multiple modules, then it belongs to $\mathcal{I}$.
2. For any $v \in \mathcal{I}$ and any module M_i, if an action $a \in M_i$ assigns to v a value calculated from multiple variables (denoted by set V_{intr}), then add all variables in $V_{intr} \setminus \mathcal{D}_{M_i}$ to $\mathcal{I}$. That is, the value assigned to an interaction variable by any action in M_i should be calculated from values of variables in interaction variables or dependency variables of the module, i.e., $\mathcal{I} \cup \mathcal{D}_{M_i}$.
3. For any variable $v \in \mathcal{D}_{M_i} \setminus \mathcal{I}$ in any module M_i, if an action assigns to v a value calculated from multiple variables (denoted by set V'_{intr}), then add all variables in $V'_{intr} \setminus \mathcal{D}_{M_i}$ to $\mathcal{I}$. That is, the value assigned to a "internal" variable of M_i by any action should be calculated from values of interaction variables or from values of dependency variables of the module, i.e., $\mathcal{I} \cup \mathcal{D}_{M_i}$.

Note that Rule 1 of this definition is conservative. Some variable x in both $\mathcal{D}_{M_i}$ and $\mathcal{D}_{M_j}$ may not convey any interaction between M_i and M_j. However, in practice such cases are rare (see the case study in §4).

The coarsening preserves interaction if: (1) all dependency variables of the target module, as well as all interaction variables remain unchanged after the coarsening; (2) all the updates of the dependency variables and interaction variables remain unchanged after the coarsening. Put it differently, only variables in $\mathcal{V}\setminus(\mathcal{I} \cup \mathcal{D}_{M_i})$ and state updates only involving such variables can be omitted during coarsening.

We use the interaction preservation rules to write coarse-grained specifications, while not affecting state-space exploration of the target module. The mixed-grained specifications help us tame state-space explosion.

B.3 Condensation

The basic idea of condensation is to merge a set of equivalent states into one state. The equivalence between states is established based on the projection to the target module M_i. After condensation of equivalent states in one trace, we can also condense equivalent traces, also based on the projection to M_i. Only after condensation, can we define the equivalence between two sets of traces in §B.4.

For specifications $S = \bigcup_{1\le i\le n} M_i$, the state s is defined as the valuation of all variables in S. State $s|_{M_i}$ is defined as the projection of s to M_i, i.e., $s|_{M_i}$ is the valuation of all variables in $\mathcal{D}_{M_i} \cup \mathcal{I}$.

For state s allowed by specification S and state $\widetilde{s}$ allowed by specification S_i, we say that s is equivalent to $\widetilde{s}$ if $s|_{M_i} = \widetilde{s}|_{M_i}$. This equivalence relation is denoted as $s \overset{M_i}{\sim} \widetilde{s}$. Similarly, we can also define the equivalence between two states s_1 and s_2 both from the same trace. We have $s_1 \overset{M_i}{\sim} s_2$ if $s_1|_{M_i} = s_2|_{M_i}$. Basically, the equivalence in both cases means that all variables in $\mathcal{D}_{M_i} \cup \mathcal{I}$ remain unchanged in two states.

For a state transition $s_1 \longrightarrow s_2$, the transition is *interesting* to M_i, if $\neg(s_1 \overset{M_i}{\sim} s_2)$. The transition is *not-interesting*, if $s_1 \overset{M_i}{\sim} s_2$. In other words, in an interesting transition, one or more variables in $\mathcal{D}_{M_i} \cup \mathcal{I}$ are updated. In a not-interesting transition, all variables in $\mathcal{D}_{M_i} \cup \mathcal{I}$ remain unchanged.

We define the *condensation* of a trace as omitting not-interesting transitions. Given a trace t allowed by S or S_i, for any transition $s_1 \longrightarrow s_2$ in t, if the transition is not-interesting, we condense the transition by merge s_1 and s_2 to a set of equivalent states $\{s_1, s_2\}$. After the condensation, a trace is the transitions from one set of equivalent states to another. Since all states merged are equivalent, they can be viewed as just one state, and the condensed trace can still be viewed as state transitions.

After the condensation of each trace, we can also condense a set of traces. For any two condensed traces allowed by some specification S, if the two states of the same index are equivalent for every index, we deem these two traces equivalent, and they are merged into a set of equivalent traces. Similarly, since all traces merged are equivalent, they can be viewed as just one trace.

B.4 Equivalence

Given that every trace is condensed and the set of traces is condensed, we can construct the equivalence relation between two sets of traces T_S and T_{S_i}.

After the condensation, any state transition is some update of state $s|_{M_i}$. Given the equivalence relation between states, the equivalence relation can also be established between two sets of traces T_S and $T_{\widetilde{S}}$. In both cases, the equivalence relations are with respect to M_i.

We define: $T_S \overset{M_i}{\sim} T_{S_i} \overset{def}{=\!=} T_S|_{M_i} = T_{S_i}|_{M_i}$. Here, by $T_S|_{M_i} = T_{S_i}|_{M_i}$, we mean that: (1) there is a bijective mapping between $t \in T_S$ and $\widetilde{t} \in T_{S_i}$; (2) for every state $s_k \in t$ and $\widetilde{s_k} \in \widetilde{t}$, $s_k \overset{M_i}{\sim} \widetilde{s_k}$.

The bijective mapping between t and $\widetilde{t}$ can be derived from the fact that the coarsening ensures interaction preservation. The mapping is constructed based on the induction on state index k in the trace.

The initial state of model checking is set the same (with projection to M_i) before and after the coarsening. Thus for any trace $t \in T_S$ and $\widetilde{t} \in T_{S_i}$, $s_0 \overset{M_i}{\sim} \widetilde{s_0}$.

By the induction hypothesis, for state $s_k \in t$, we have exactly one corresponding state $\widetilde{s_k}$ with $s_k \overset{M_i}{\sim} \widetilde{s_k}$. Now consider the transition $s_k \longrightarrow s_{k+1}$. After the trace condensation, s_k and s_{k+1} are not equivalent, and $s_k \longrightarrow s_{k+1}$ must involve update of one or more variables in $\mathcal{D}_{M_i} \cup \mathcal{I}$.

According to interaction preservation rule (§B.2), the variables in $\mathcal{D}_{M_i} \cup \mathcal{I}$ and updates of these variables in any action remain unchanged after the coarsening. So we have the same updates on the same set of variables, i.e. variables in $\mathcal{D}_{M_i} \cup \mathcal{I}$, for s_k and $\widetilde{s_k}$.

Note that each state transition is deterministic. This is ensured by the fact that in all our specifications S and S_i, any state updates in any action is deterministic. The same updates on the same set of variables deterministically produce one unique next state, given that we only consider the projection to M_i. That is, we are only concerned of state updates of the target module M_i and ignore other information in the states and the state transitions. Thus, s_k and $\widetilde{s}_k$ both have their unique successors s_{k+1} and $\widetilde{s}_{k+1}$ respectively, and $s_{k+1} \overset{M_i}{\sim} \widetilde{s}_{k+1}$.

By induction, we have the desired mapping between traces in $T_S|_{M_i}$ and $T_{S_i}|_{M_i}$. This gives us that $T_S \overset{M_i}{\sim} T_{S_i}$, which ensures the safety of model checking after the coarsening. □

References

[1] ZOOKEEPER-2355. Ephemeral node is never deleted if follower fails while reading the proposal packet. https://issues.apache.org/jira/browse/ZOOKEEPER-2355, 2016.

[2] ZOOKEEPER-2678. Large databases take a long time to regain a quorum. https://issues.apache.org/jira/browse/ZOOKEEPER-2678, 2017.

[3] ZOOKEEPER-2845. Data inconsistency issue due to retaining database in leader election. https://issues.apache.org/jira/browse/ZOOKEEPER-2845, 2017.

[4] ZOOKEEPER-3023. Assertion fails when follower's history is not in sync with the leader's initial history after leader receives its ACK for NEWLEADER. https://issues.apache.org/jira/browse/ZOOKEEPER-3023, 2018.

[5] ZOOKEEPER-3104. Potential data inconsistency due to NEWLEADER packet being sent too early during SNAP sync. https://issues.apache.org/jira/browse/ZOOKEEPER-3104, 2018.

[6] ZOOKEEPER-3642. Data inconsistency when the leader crashes right after sending SNAP sync. https://issues.apache.org/jira/browse/ZOOKEEPER-3642, 2019.

[7] ZOOKEEPER-3911. Data inconsistency caused by DIFF sync uncommitted log. https://issues.apache.org/jira/browse/ZOOKEEPER-3911, 2020.

[8] ZOOKEEPER-4394. Learner.syncWithLeader got NullPointerException. https://issues.apache.org/jira/browse/ZOOKEEPER-4394, 2021.

[9] Azure Cosmos TLA+ specifications. https://github.com/Azure/azure-cosmos-tla, 2022.

[10] TLA+ in TiDB. https://github.com/pingcap/tla-plus, 2022.

[11] ZOOKEEPER-4541. Ephemeral znode owned by closed session visible in 1 of 3 servers. https://issues.apache.org/jira/browse/ZOOKEEPER-4541, 2022.

[12] ZOOKEEPER-4643. Committed transactions are improperly truncated when follower crashes right after updating currentEpoch. https://issues.apache.org/jira/browse/ZOOKEEPER-4643, 2022.

[13] ZOOKEEPER-4646. Transaction loss when followers crash after replying ACK of NEWLEADER before logging transactions to disk. https://issues.apache.org/jira/browse/ZOOKEEPER-4646, 2022.

[14] Trace validation for the Raft consensus algorithm in etcd implementation. https://github.com/etcd-io/raft/pull/113, 2023.

[15] ZOOKEEPER-4685. Unnecessary system unavailability due to leader shutdown when follower sends ACK of PROPOSAL before ACK of NEWLEADER. https://issues.apache.org/jira/browse/ZOOKEEPER-4685, 2023.

[16] ZOOKEEPER-4712. Follower shutdown() does not correctly shutdown SyncProcessor, which leads to data inconsistency. https://issues.apache.org/jira/browse/ZOOKEEPER-4712, 2023.

[17] Apache ZooKeeper. https://zookeeper.apache.org/, 2024.

[18] AspectJ. https://eclipse.dev/aspectj/, 2024.

[19] Jepsen. https://jepsen.io/, 2024.

[20] Microsoft CCF TLA+ Specifications. https://github.com/microsoft/CCF/tree/b10483af676354e21c19432099fcf43bdb6201ee/tla, 2024.

[21] MongoDB TLA+/PlusCal Specifications. https://github.com/mongodb/mongo/tree/r7.3.2/src/mongo/tla_plus, 2024.

[22] Proptest documentation. https://altsysrq.github.io/proptest-book/, 2024.

[23] The Java Remote Method Invocation API (Java RMI). https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html, 2024.

[24] TLA+ Specification for etcd Raft. https://github.com/etcd-io/raft/tree/9ee2dd30d6a67a64a62ec8258bb33316c6f60283/tla, 2024.

[25] TLC and TLA+ Toolbox. https://github.com/tlaplus/tlaplus, 2024.

[26] ZOOKEEPER-4785. Transaction loss due to race condition in Learner.syncWithLeader() during DIFF sync. https://issues.apache.org/jira/browse/ZOOKEEPER-4785, 2024.

[27] ZooKeeper Use Cases. https://zookeeper.apache.org/doc/r3.9.1/zookeeperUseCases.html, 2024.

[28] ZooKeeper's learner code. https://github.com/apache/zookeeper/blob/release-3.9.1/zookeeper-server/src/main/java/org/apache/zookeeper/server/quorum/Learner.java, 2024.

[29] Abadi, M., and Lamport, L. Composing Specifications. *ACM Transactions on Programming Languages and Systems 15*, 1 (Jan. 1993), 73–132.

[30] Abadi, M., and Lamport, L. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems 17*, 3 (May 1995), 507–535.

[31] Agrawal, A., and Policzer, Z. TLA+ @ LinkedIn: Ambry & Venice. In *TLA+ Conference* (Apr. 2024). https://youtu.be/Jz0J5N77QKk.

[32] Almeida, J. B., Barbosa, M., Barthe, G., Campagna, M., Cohen, E., Gregoire, B., Pereira, V., Portela, B., Strub, P.-Y., and Tasiran, S. A Machine-Checked Proof of Security for AWS Key Management Service. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)* (Nov. 2019).

[33] Alvaro, P., Rosen, J., and Hellerstein, J. M. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)* (May 2015).

[34] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., and Rosenthal, C. Chaos Engineering. *IEEE Software 33*, 3 (Mar. 2016), 35–41.

[35] Bornholt, J., Joshi, R., Astrauskas, V., Cully, B., Kragl, B., Markle, S., Sauri, K., Schleit, D., Slatton, G., Tasiran, S., Van Geffen, J., and Warfield, A. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)* (Oct. 2021).

[36] Brooker, M. Fifteen Years of Formal Methods at AWS. In *TLA+ Conference* (Apr. 2024). https://youtu.be/HxP4wi4DhA0.

[37] Chen, H., Dou, W., Wang, D., and Qin, F. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering (ASE'20)* (Sept. 2020).

[38] Chen, Y., Sun, X., Nath, S., Yang, Z., and Xu, T. "Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker". In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)* (Apr. 2023).

[39] Chou, D., Xu, T., Veeraraghavan, K., Newell, A., Margulis, S., Xiao, L., Ruiz, P. M., Meza, J., Ha, K., Padmanabha, S., Cole, K., and Perelman, D. Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[40] Cirstea, H., Kuppe, M. A., Loillier, B., and Merz, S. Validating Traces of Distributed Programs Against TLA+ Specifications. *arXiv preprint arXiv:2404.16075* (Apr. 2024).

[41] Clarke, E. M., Long, D. E., and McMillan, K. L. Compositional Model Checking. In *Proceedings of the 4th Annual Symposium on*

Logic in Computer Science (LICS'89) (June 1989).

[42] Davis, A. J. J., Hirschhorn, M., and Schvimer, J. eXtreme Modelling in Practice. In *Proceedings of the VLDB Endowment (VLDB'20)* (May 2020).

[43] Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., and Zufferey, D. P: Safe Asynchronous Event-Driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)* (June 2013).

[44] Elmas, T., Tasiran, S., and Qadeer, S. VYRD: VerifYing Concurrent Programs by Runtime Refinement-Violation Detection. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)* (June 2005).

[45] Ganesan, A., Alagappan, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)* (Feb. 2018).

[46] Gu, J. T., Sun, X., Zhang, W., Jiang, Y., Wang, C., Vaziri, M., Legunsen, O., and Xu, T. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP'23)* (Oct. 2023).

[47] Gu, X., Cao, W., Zhu, Y., Song, X., Huang, Y., and Ma, X. Compositional Model Checking of Consensus Protocols via Interaction-Preserving Abstraction. In *Proceedings of the 41st International Symposium on Reliable Distributed Systems (SRDS'22)* (Sept. 2022).

[48] Guerraoui, R., and Yabandeh, M. Model Checking a Networked System Without the Network. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (Mar. 2011).

[49] Gunawi, H. S., Do, T., Joshi, P., Alvaro, P., Hellerstein, J. M., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Sen, K., and Borthakur, D. Fate and Destini: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (Mar. 2011).

[50] Guo, H., Wu, M., Zhou, L., Hu, G., Yang, J., and Zhang, L. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (Oct. 2011).

[51] Hackett, F., Hosseini, S., Costa, R., Do, M., and Beschastnikh, I. Compiling Distributed System Models with PGo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)* (Jan. 2023).

[52] Hackett, F., Rowe, J., and Kuppe, M. A. Understanding Inconsistency in Azure Cosmos DB with TLA+. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'23)* (May 2023).

[53] Hance, T., Heule, M., Martins, R., and Parno, B. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)* (Apr. 2021).

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

[55] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[56] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (Mar. 2011).

[57] Honoré, W., Kim, J., Shin, J.-Y., and Shao, Z. Much ADO about Failures: A Fault-Aware Model for Compositional Verification of Strongly Consistent Distributed Systems. In *Proceedings of the 2021 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'21)* (Oct. 2021).

[58] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)* (June 2010).

[59] Jonsson, B. Compositional Specification and Verification of Distributed Systems. *ACM Transactions on Programming Languages and Systems 16*, 2 (Mar. 1994), 259–303.

[60] Junqueira, F. P., Reed, B. C., and Serafini, M. Dissecting Zab. Tech. Rep. YL-2010-007, Yahoo! Research, Dec. 2010.

[61] Junqueira, F. P., Reed, B. C., and Serafini, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'11)* (June 2011).

[62] Killian, C., Anderson, J. W., Jhala, R., and Vahdat, A. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)* (Apr. 2007).

[63] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)* (June 2007).

[64] Kim, B. H., Kim, T., and Lie, D. Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC'22)* (July 2022).

[65] Kuppe, M. A. Validating System Executions with the TLA+ Tools. In *TLA+ Conference* (Apr. 2024). https://youtu.be/NZmON-XmrkI.

[66] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Aug. 2002.

[67] Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., and Hawblitzel, C. Verus: Verifying Rust Programs Using Linear Ghost Types. In *Proceedings of 2023 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'23)* (Apr. 2023).

[68] Lauterburg, S., Dotta, M., Marinov, D., and Agha, G. A Framework for State-Space Exploration of Java-Based Actor Programs. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)* (Nov. 2009).

[69] Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J. F., and Gunawi, H. S. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[70] Leesatapornwongsa, T., Lukman, J. F., Lu, S., and Gunawi, H. S. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the 21st International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'16)* (Apr. 2016).

[71] Leino, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)* (Apr. 2010).

[72] Liu, H., Li, G., Lukman, J. F., Li, J., Lu, S., Gunawi, H. S., and Tian, C. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the 22nd International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'17)* (Apr. 2017).

[73] Liu, H., Wang, X., Li, G., Lu, S., Ye, F., and Tian, C. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings*

of the 23rd International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'18) (Mar. 2018).

[74] Loncaric, C. Reverse-Engineering with TLA+ at Oracle. In *TLA+ Conference* (Apr. 2024). https://youtu.be/dGBSeagCAxw.

[75] Lu, J., Liu, C., Li, L., Feng, X., Tan, F., Yang, J., and You, L. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'19)* (Oct. 2019).

[76] Lukman, J. F., Ke, H., Stuardo, C. A., Suminto, R. O., Kurniawan, D. H., Simon, D., Priambada, S., Tian, C., Ye, F., Leesatapornwongsa, T., Gupta, A., Lu, S., and Gunawi, H. S. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)* (Mar. 2019).

[77] Ma, H., Ahmad, H., Goel, A., Goldweber, E., Jeannin, J.-B., Kapritsos, M., and Kasikci, B. Sift: Using Refinement-guided Automation to Verify Complex Distributed Systems. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC'22)* (July 2022).

[78] Ma, H., Goel, A., Jeannin, J.-B., Kapritsos, M., Kasikci, B., and Sakallah, K. A. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[79] Marić, O., Sprenger, C., and Basin, D. Cutoff Bounds for Consensus Algorithms. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV'17)* (July 2017).

[80] McMillan, K. L., and Padon, O. Ivy: A Multi-Modal Verification Tool for Distributed Algorithms. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV'20)* (July 2020).

[81] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. How Amazon Web Services Uses Formal Methods. *Communications of the ACM 58*, 4 (Mar. 2015), 66–73.

[82] Ouyang, L., Huang, Y., Huang, B., and Ma, X. Leveraging TLA+ Specifications to Improve the Reliability of the ZooKeeper Coordination Service. In *Proceedings of the 9th International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA'23)* (Nov. 2023).

[83] Ozkan, B. K., Majumdar, R., Niksic, F., Befrouei, M. T., and Weissenbacher, G. Randomized Testing of Distributed Systems with Probabilistic Guarantees. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'18)* (Oct. 2018).

[84] Padon, O., McMillan, K. L., Panda, A., Sagiv, M., and Shoham, S. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)* (June 2016).

[85] Reed, B., and Junqueira, F. P. A simple totally ordered broadcast protocol. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS'08)* (Sept. 2008).

[86] Reed, B., Junqueira, F. P., and Han, M. Zab1.0. https://cwiki.apache.org/confluence/display/ZOOKEEPER/Zab1.0, May 2021.

[87] Saissi, H., Bokor, P., Muftuoglu, C. A., Suri, N., and Serafini, M. Efficient Verification of Distributed Protocols Using Stateful Model Checking. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems (SRDS'13)* (Sept. 2013).

[88] Sergey, I., Wilcox, J. R., and Tatlock, Z. Programming and Proving with Distributed Protocols. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[89] Sharma, U., Jung, R., Tassarotti, J., Kaashoek, F., and Zeldovich, N. Grove: A Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)* (Oct. 2023).

[90] Simsa, J., Bryant, R., and Gibson, G. dBug: Systematic Evaluation of Distributed Systems. In *Proceedings of the 5th International Conference on Systems Software Verification (SSV'10)* (Oct. 2010).

[91] Sun, X., Luo, W., Gu, J. T., Ganesan, A., Alagappan, R., Gasch, M., Suresh, L., and Xu, T. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[92] Sun, X., Ma, W., Gu, J. T., Ma, Z., Chajed, T., Howell, J., Lattuada, A., Padon, O., Suresh, L., Szekeres, A., and Xu, T. Anvil: Verifying Liveness of Cluster Management Controllers. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)* (July 2024).

[93] Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., and Karl, R. Holistic Configuration Management at Facebook. In *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)* (Oct. 2015).

[94] Tang, R., Sun, X., Huang, Y., Wei, Y., Ouyang, L., and Ma, X. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys'24)* (Apr. 2024).

[95] Tasiran, S., Yu, Y., and Batson, B. Using a Formal Specification and a Model Checker to Monitor and Direct Simulation. In *Proceedings of the 40th Annual Design Automation Conference (DAC'03)* (June 2003).

[96] Tseitlin, A. The Antifragile Organization. *Communications of the ACM 56*, 8 (Aug. 2013), 40–44.

[97] Wang, D., Dou, W., Gao, Y., Wu, C., Wei, J., and Huang, T. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)* (May 2023).

[98] Wilcox, J. R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M. D., and Anderson, T. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)* (June 2015).

[99] Wu, H., Pan, J., and Huang, P. Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)* (Apr. 2024).

[100] Yabandeh, M., Knezevic, N., Kostic, D., and Kuncak, V. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)* (Apr. 2009).

[101] Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., and Zhou, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)* (Apr. 2009).

[102] Yao, J., Tao, R., Gu, R., and Nieh, J. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[103] Yao, J., Tao, R., Gu, R., and Nieh, J. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. In *Proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'24)* (Jan. 2024).

[104] Yao, J., Tao, R., Gu, R., Nieh, J., Jana, S., and Ryan, G. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)* (July 2021).

[105] Yuan, X., and Yang, J. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the 25th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'20)* (Mar. 2018).

[106] Zhou, S. How We Designed and Model-Checked MongoDB Reconfiguration Protocol. In *TLA+ Conference* (Apr. 2024). https://youtu.be/-eAktIBUhHA.