Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models

Beom Heyn Kim, *Samsung Research and University of Toronto;* Taesoo Kim, *Samsung Research and Georgia Institute of Technology;* David Lie, *University of Toronto*

https://www.usenix.org/conference/atc22/presentation/kim-beom-heyn

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models

Beom Heyn Kim[§†], Taesoo Kim[§‡], and David Lie[†]
[§]Samsung Research [†]University of Toronto [‡]Georgia Institute of Technology
{beomheyn.kim,tsgates.kim}@samsung.com, lie@eecg.toronto.edu

Abstract

While there exist many consistency models for distributed systems, most of those models seek to provide the basic guarantee of convergence: given enough time and no further inputs, all replicas in the system should eventually converge to the same state. However, because of *Convergence Failure Bugs* (CFBs), many distributed systems do not provide even this basic guarantee. The violation of the convergence property can be crucial to safety-critical applications collectively working together with a shared distributed system. Indeed, many CFBs are reported as major issues by developers. Our key insight is that CFBs are caused by divergence, or differences between the state of replicas, and that a focused exploration of divergence states can reveal bugs in the convergence logic of real distributed systems while avoiding state explosion. Based on this insight, we have designed and implemented Modulo, the first Model-Based Testing tool using *Divergence Resync Models* (DRMs) to systematically explore divergence and convergence in real distributed systems. Modulo uses DRMs to explore an abstract state machine of the system and derive schedules, the intermediate representation of test cases, which are then translated into test inputs and injected into systems under test (SUTs). We ran Modulo to check ZooKeeper, MongoDB, and Redis and found 11 bugs (including 6 previously unknown ones)

1 Introduction

The emergence of cloud-scale applications has driven the need for distributed storage systems to support them. To provide availability and scalability, those systems replicate data across several replicas, which may be distributed in a single datacenter or even globally across several datacenters [1–5, 32, 39, 50, 57, 62]. To tolerate network partitions and delays, many of these systems adopt weaker-consistency guarantees [58, 59], allowing them to replicate data asynchronously. This means that clients connected to different replicas may observe different states of the data. The exact order and delay of concurrent operations on replicated data is governed by a "consistency model," which attempts to strike a balance between intuitive behavior (favoring stronger consistency guarantees) and scalability and partition tolerance (favoring weaker guarantees).

Regardless of these differences, most of consistency models in practice guarantee a common property, which is that given enough time and no further modifications to the data, all replicas will eventually arrive at the same contents for the data—something we call the *convergence guarantee* also known as eventual consistency. However, distributed systems are inherently designed to be *temporarily inconsistent* so that they may continue to respond to requests, even as they *converge* to a consistent state by replicating data among the replicas. We call this temporary inconsistency *divergence*. Yet, divergence can cause more than temporary inconsistency in the presence of failures. Divergence in the presence of failures may also lead to *conflicts*, where different replicas have incompatible states, which can only be resolved by truncating or removing data from one or more of the replicas. While it is not the sole cause, we find that a major cause of systems failing to converge is defects in the convergence logic after the failure recovery. Such bugs leading to convergence failures are named *Convergence Failure Bugs* (CFBs).

Looking into bug databases of a couple of systems for the period from 2010 to 2017, we found about 10 bugs that are already reported and fixed by developers and external users [23, 31, 33, 34, 51, 55, 56, 60, 64]. They are all marked by developers as either "Blocker", "Critical" or "Major" in terms of severity. This demonstrates that CFBs are perceived by developers as real, prevalent and important bugs to find and fix. In some case, the convergence failure is noticed by developers as visible to clients [31], which can directly cause clients to make incorrect decisions leading to serious consequences. Thus, a convergence failure can be crucial to safety-critical applications collectively working together through a shared distributed system.

To exercise distributed systems' convergence logic, more divergence than would naturally occur during regular use needs to be generated. This paper presents *Modulo*, the first Model-Based Testing tool that systematically explores differ-

ent divergence states by alternately injecting events that cause divergence and convergence into the real distributed systems.

Modulo overcomes limitations in previous solutions for detecting CFBs in distributed systems. On one hand, distributed systems model-checkers [27, 29, 35, 41, 42, 44, 52, 61] aim to provide formal verification of a distributed system, and as such must ensure that they exhaustively explore the state space of the system under test (SUT). To achieve exhaustive exploration, they must tightly control all nondeterministic events so as to drive the SUT through all possible states. Unfortunately, tightly controlling all events leads to the well-known "state-explosion problem," as the number of states grows exponentially with the number of events that are controlled. To reduce the severity of state-explosion, a smart and insightful abstraction of target behavior is needed. None of existing model-checkers has explored the state-space consisting of divergence and convergence.

On the other hand, random testing approaches, such as Jepsen [38], do not aim for formal verification but rather simply to find bugs, and thus they need not exhaustively explore all states. This frees them from having to control all nondeterministic events. Instead, random testing approaches inject a targeted set of external events (randomly of course) and, rather than controlling all other events, allow the SUT to randomly visit states depending on how events interleave naturally during execution. Random testing approaches typically do not record the states explored, so they cannot provide any guarantee or measure of state-space coverage. Moreover, because they do not know which events are important, they may not be able to provide a sequence of inputs that can reliably reproduce the bug.

Modulo's key contributions stem from our observation that *many CFBs arise from flaws in the convergence logic of distributed systems, and are orthogonal to other functions of the system*. Thus, CFBs can often be reproduced purely by partially controlling only the few events that lead to convergence and divergence. This partial control allows Modulo to significantly reduce the severity of the state explosion problem, while still enabling it to deeply explore different divergence states of the SUT.

Different distributed systems have different techniques to converge replicas after a failure. To abstract these differences so that it can generalize across different systems, Modulo introduces *Divergence Resync Models* (DRMs), which consist of an *Abstract Execution Model* (AEM) and a *Concrete Execution Model* (CEM). The AEM describes abstract conditions for convergence and divergence events. For example, systems like ZooKeeper and MongoDB require quorum before they can accept client requests that could cause divergence between replicas. The AEM for these systems thus specifies the conditions under which the systems quorum will have been achieved (i.e., the majority of replicas are available). The CEM maps the AEM conditions, as well as divergence and convergence events, to API calls for a specific SUT. Modulo uses the AEM to generate schedules of abstract events that alternate between divergence and convergence and the CEM to execute these schedules on the SUT to search the convergence code of the system for CFBs.

We ran Modulo on ZooKeeper, MongoDB, and Redis as SUTs and found 11 CFBs, including 6 new ones that had not been found before. For each of these bugs, Modulo provides a schedule of inputs that deterministically triggers the bug. To find these CFBs, we used 5 DRMs—1 for ZooKeeper, 1 for MongoDB, and 3 for Redis, which range from 72-782 lines of code in size.

We made the following novel contributions:

- As far as we know, Modulo is the first systematic test generation system, specifically designed to discover CFBs.
- We introduce the concept of Divergence Resync Models (DRMs) that inject events specifically designed to elicit and discover the existence of CFBs.
- We design, implement and evaluate Modulo, a system that uses DRMs to find CFBs in real distributed systems.
- We perform an empirical study to demonstrate the effectiveness of the proposed approach by integrating the prototype to 3 mature open-source distributed systems: ZooKeeper, MongoDB, and Redis. Modulo was able to find several critical CFBs in them.

§2 gives the overview of divergence and convergence, the core concepts of Modulo and DRMs, and provides an example CFB that Modulo can find. §3 describes the architecture of Modulo and the 5 DRMs we use in this study. We then document our experience and empirical results of applying Modulo to mature open-source distributed systems in §4. Subsequently, we present further discussion on Modulo compared to previous proposals in §5 and discuss related work in §6. Finally, we draw our conclusions in §7.

2 Overview

2.1 System Model

We model a distributed storage system as a set of replicas, each of which is a key-value store. In an idealized system, the storage system consists of one replica, and each write would transition the key-value store in the system from one state to the next. However, in the distributed implementation of the system, each write is applied to one of the replicas, and then the distributed storage system asynchronously replicates the write to the remaining replicas until every replica converges to the same state. Thus, at any given time, there can be several replicas that differ from our idealized single replica. Our target system should manage updates and resolve conflicts to maintain the convergence of replicas. That is, our proposed

approach is not designed to test those systems allowing multi-master updates with gossip protocols where the convergence property is not guaranteed.

Our approach injects failures into the system-under-test to drive it into corner cases. Failures are erroneous states where synchronization between replicas is interrupted, so convergence does not occur without reverting to the normal state via failure recoveries. Failures may be caused by several reasons, such as crash of replicas, suspension of replicas, and links failures between replicas. After recovering from failures, convergence should occur and replicas must have identical structure in terms of write sequences contained in their log. Regarding the convergence property, we focus on those systems that are partially synchronous, although there is nothing stopping us from applying Modulo to eventually consistent systems such as Cassandra, which is left as a future work. Quorum-based systems we tested with Modulo require a quorum to elect a leader, but may ingest writes expecting failures will be recovered soon. In contrast, Redis does not use leader election, so it does not require a quorum to start servicing clients. It lets the sync source to replicate any changes down to the sync targets recursively.

2.2 Divergence and Convergence

We formally define *divergence* as the total number of writes that need to be applied across all replicas to make their key-value stores equal to the single idealized replica. *Convergence* is simply the complement of divergence (i.e., $convergence = -divergence$), where the replicas are said to be converged when divergence is zero. *Resynchronization* (resync) is an operation implemented by distributed storage systems to achieve convergence after a failed replica recovers. Resync reduces divergence by replicating writes from a replica to the recovered replica and may implement *conflict resolution* to eliminate write operations that prevent the replicas from achieving convergence. Conflict resolution is particularly complex as it usually results in data loss, which storage systems seek to avoid unless absolutely necessary.

Divergence occurs in the natural course of the operation of a distributed system as writes are applied to replicas. However, under normal circumstances, the amount of divergence is usually small, as systems aim to replicate writes fairly quickly, subject to standard networking and processing delays. However, failures may further increase divergence. For instance, replica crashes and network outages can prevent replicas from replicating operations for an extended period of time. This will trigger defects related to assumptions about the resources needed to track outstanding operations or about the length of time that replicas may be unavailable. Repeated failures may result in replicas changing leader or master roles, which can result in conflicts, allowing Modulo to exercise the conflict resolution logic of distributed systems.

A simple example illustrating the concept of divergence and convergence is given in Figure 1. The divergence example starts in an initial state where A is the leader replica and B and C are non-leader replicas. At time ❶ , a write of v0 to k, $W(k, v0)$, is sent by a client to A, A applies the operation and then replicates it to B and C. At time ❷ , B fails and, another write, $W(k, v1)$, is sent to A. Only A and C can apply the write, thus there is some amount of temporary divergence among replicas. At time ❸ , C fails, and another write, $W(k, v2)$, is sent to A. Only A can apply the write, resulting in even more divergence among the 3 replicas. Convergence can simply be regarded as the decrease of divergence in the system. At time ❹ , B recovers from the failure and rejoins the distributed system. Most distributed systems implement resync procedures that attempt to bring B up to date with the most recent state on the other replicas. Thus, B can converge with A by replicating $W(k, v2)$. At time ❺ , C recovers and resyncs with A by replicating $W(k, v2)$, resulting in full convergence among the 3 replicas. Now, there is no divergence in the system. (i.e., all replicas have the same key-value stores).

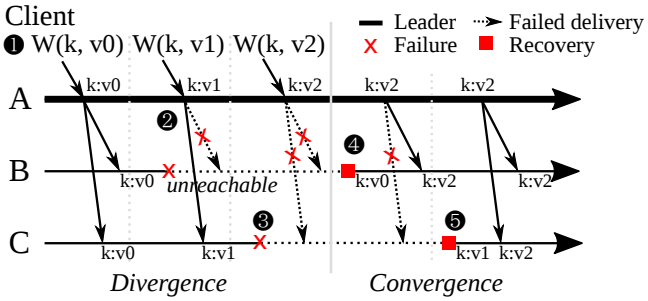

Figure 1: **Divergence and convergence.** Replicas A, B and C diverge their states upon failures of B and C. Since the leader replica A is alive, the latest state of A can be replicated to replicas B and C upon their recovery, resulting in the state *convergence*.

2.3 An Example Bug

To illustrate the type of bugs Modulo will find, we give an example of a new bug that Modulo discovered in ZooKeeper version 3.4.11 [8]. ZooKeeper requires a quorum of replicas to be online to operate, where quorum is defined as more than one-half the total number of replicas. In each epoch, which is a predefined period of time, the replicas in the quorum elect one of them as the "leader." All write operations are serialized through the elected leader, and all other replicas (called "followers") replicate operations from the leader.

ZooKeeper replicas employ 2 mechanisms to save data so that it can be recovered after a crash. First, the replicas use a write-ahead transaction log that can be replayed after a failure to recover the state of data that did not properly persist. Transactions are appended into the log as they get committed. Because all changes caused by transactions are

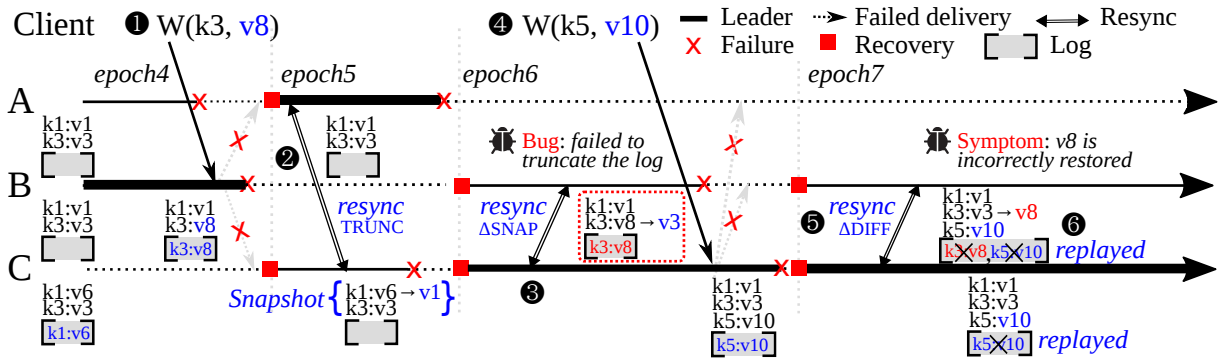

Figure 2: **Running example (ZooKeeper Bug#1).** The replica B fails to truncate the past log (`k3: v8`) upon recovery with the SNAP resync (see, §2.3). To observe this bug, another failure and recovery step is required right after the bug is triggered (❻). Modulo successfully formulated the exact sequences of steps to trigger this previously unknown CFB.

sequentially logged, replaying transactions in the log will restore the state of a replica. Second, replicas periodically clear the transaction log and persist a snapshot of in-memory data to disk, which can then be reloaded into memory after a failure. Taking a snapshot of memory is considerably slower than writing transactions into the write-ahead log as they occur, so ZooKeeper only takes snapshots after the transaction log has grown to some point or after resync following a failure depending on the resync logic. For example, in ZooKeeper 3.4.11, taking snapshot occurs on followers if they resync with a new leader via SNAP or by sending a truncation request, which are further explained in following paragraphs. The example is replica C taking a snapshot after the resync at time ❷ in Figure 2.

ZooKeeper uses 2 mechanisms to resync replicas after replicas have recovered from a failure. One mechanism is DIFF resync, where another replica transfers all missing operations from its transaction log to the recovered replica. The other mechanism is SNAP resync, where another replica sends its entire key-value store to the recovered replica. ZooKeeper picks DIFF resync if the leader's log contains all transactions required for resync. However, this can lead to problems as old log entries may be purged by an earlier snapshot. If ZooKeeper's resync logic determines the leader does not have all required transactions in its log, then the SNAP mechanism will be selected. For example, in ZooKeeper version 3.4.11, when followers resync with the leader that does not have a log containing entries that are newer than its snapshot, the SNAP mechanism is used (e.g., SNAP resync at time ❸ in Figure 2).

The base case that can happen during the resync is replicating operations that have not been replicated to those replicas recovered from a failure. Moreover, it may be necessary for a recovering replica to *truncate* its local write-ahead log and remove conflicting operations. During resync, a leader sends a truncation request (TRUNC) to a follower. Conflicting operations may exist if one replica had been the leader and committed some operations that had not yet been replicated to other replicas before failing, and another replica is subsequently elected as the new leader and commits another set of operations, resulting in two conflicting sets of operations.

We illustrate the example bug in Figure 2. Suppose we initially have replicas A, B and C and the epoch is at 4. Both A and B have the same set of key-value pairs $k1:v1$, $k2:v2$, $k3:v3$, $k4:v4$ and $k5:v5$. Also, their log contains entries for all writes creating the key-value set. C has the same key-value set except for $k1:v6$ and its log additionally contains the entry for the write, $W(k1,v6)$. Currently, A and B are up and C is down. B is the leader of the epoch 4.

At time ❶ , A crashes and a write, $W(k3,v8)$, is made, which B accepts and commits [1]. Then, B crashes. At time ❷ , A and C restart and resync using TRUNC. C restores its key-value set by replaying its truncated log and takes a snapshot. Note that neither A nor C has seen the write, $W(k3,v8)$, yet at this point. Then, A and C crash and B and C restart. C becomes the new leader. At time ❸ , because C has no log entry newer than its snapshot, C resync with B using the SNAP mechanism. Yet, C does not send a truncation request to B, so B accepts and restores using the C's snapshot but fails to truncate the write, $W(k3,v8)$, from its log, which is the root cause of the bug. However, at this point, the bug is not apparent yet. At time ❹ , B crashes and another write, $W(k5,v10)$, is committed on C. Then, C crashes and B and C restart. Now, C becomes the leader. At time ❺ , B and C use DIFF resync to replicate $W(k5,v10)$ from C to B. At time ❻ , B replays its log and restores $k3:v8$, while C still has $k3:v3$. Because of the failure to truncate the log entry for $k3:v8$ on B, the replicas believe they have converged when in fact they have not, and the system fails to reach convergence.

This example illustrates several key features of the CFBs that Modulo is designed to discover. First, a long series of very specific steps is required to trigger the bug—far larger than are likely to happen simply due to randomized stress testing. The series of steps is essentially the alternating sequence of

[1] $W(k2,v7)$ injected at the epoch 3 and $W(k4,v9)$ injected at the epoch 5 are not shown in the figure.

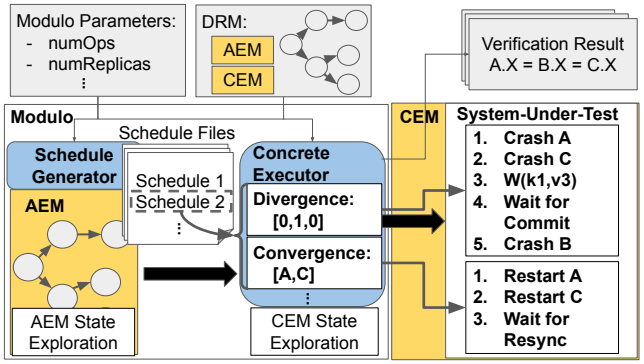

Figure 3: Modulo Architecture. Gray boxes are input and output of Modulo. Blue boxes are Modulo components. Yellow boxes are DRM components.

convergence and divergence that causes the CFB to surface. Modulo specifically targets the generation of such sequences using DRMs, which we discuss in Section 3. Second, because the sequence is very deep, it would be very difficult to find such bugs through a naïve search of the entire state space of all the replicas. However, the root cause and nature of the bug make it orthogonal to the detailed internal state of the replicas, which can be influenced by internal events such as thread interleaving and the order of lock acquisitions. Instead, it is the generation of divergence and convergence events between the replicas in ZooKeeper that triggers the bugs, which are the events that Modulo seeks to explore. By focusing only on controlling events related to divergence and convergence, such as replica failure and recovery, Modulo is able to explore deep sequences such as these without being constrained by state explosion.

3 Modulo

As shown in Figure 3, Modulo consists of 2 core components: a schedule generator and a concrete executor. To use Modulo, the user provides a DRM and 2 parameters: *numReplicas*, which indicates the number of replicas, and *numOps*, which indicates the total number of writes that will be applied to the SUT. Modulo then uses the DRM to produce a schedule of divergence transitions ($\mathcal{D}\rightarrow$) and convergence transitions ($\mathcal{C}\rightarrow$), which are then executed by the concrete executor on the SUT. $\mathcal{D}\rightarrow$ cause more divergence in a system while $\mathcal{C}\rightarrow$ cause convergence among available replicas. After executing each schedule, the concrete executor waits for the SUT to be quiescent after pre-configured time duration. Then, it checks whether all replicas in the SUT have converged (i.e., have identical state) by reading values of each key and compare those across replicas. Finally, the result is recorded in a file for more detailed analysis.

A DRM contains 2 subcomponents: an Abstract Execution Model (AEM) and a Concrete Execution Model (CEM). The AEM describes the conditions under which $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ may take place. For example, an SUT may require a quorum of replicas to be available before write operations will be accepted and divergence may occur, which would be specified in the AEM. AEMs describe such requirements abstractly in terms of the AEM state, given in Table 1, such that a single AEM may be used to test multiple SUTs. For example, both ZooKeeper and MongoDB require quorums, so an AEM that models quorums is used to test both, while an AEM that does not model quorums may be needed to test systems that do not rely on quorum. The output of the schedule generator in Modulo is a set of schedule files, which are consumed by the concrete executor. We describe this further in §3.3. The concrete executor uses the CEMs to map $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ in the schedules to concrete operations that drive a SUT into the states dictated by the schedule. As such, CEMs are necessarily specific to the SUT.

An AEM specifies a finite state machine of the system-under-test and the CEM translates transitions in the AEM into corresponding transitions on the system-under-test. The user is required to abstract away unnecessary details in the AEM to limit the state space being tested to interesting states. The relationship between the AEM and the CEM can thus be viewed as the AEM specifying the state space to test and the CEM translating tests specified by that state space into concrete tests to run on the actual SUT. Modulo exhaustively explores the AEM's finite state machine. Failures are detected directly on the SUT when it fails to converge after a certain amount of time. There are differences in the DRMs to test each SUT differently. For instance, Redis's DRMs need a way to model the link failures and recoveries between pairs of replicas, different from ZooKeeper's and MongoDB's DRM.

We describe the most generic form of AEMs and CEMs below, which can be flexibly extended to more sophisticated models. Table 1 and 2 are for the baseline AEM used for ZooKeeper and MongoDB. Our baseline AEM abstract away the role of replicas, but the leader is distinguished during concrete execution of CEM to find out to which replica a write should be injected. For more complex AEMs like the one used for Redis, we need to extend states and transitions of AEM to model the network link failures.

3.1 Abstract Execution Model

The AEM specifies a state machine, whose state space Modulo explores to produce the schedule of $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$. At each state, Modulo systematically performs $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ depending on whether the guard conditions specified in the AEM are met or not. The guard conditions are boolean functions over the state of the AEM. During $\mathcal{D}\rightarrow$, Modulo simulates a client that sends zero or more write operations from the sequence of write operations $W : \langle w_n | n \in \{0..numOps\}\rangle$, where *numOps* is the parameter specified as part of the test configuration. The term *replicaState* is a vector of length

Variable	Description
replicaState	List of non-negative integer values. State of all replicas that defines the latest write operation applied
onlineStatus	List of boolean values. Status of the replica indicating if the replica is online or offline

Table 1: AEM State Description.

Transition	Description
convergence	**replicaState:** set each online replica's write index to the latest write that the replica can resynchronize to **onlineStatus:** set one or more replicas to online
divergence	**replicaState:** increase replica's write index based on number of writes applied **onlineStatus:** set zero or more replicas to offline

Table 2: AEM Transition Description.

numReplicas, with each element indicating the index of the latest write operation in W that a particular replica has observed. The term *onlineStatus* is also a vector of length *numReplicas*, which stores the status of each replica as either *online*, meaning that the replica is available, or *offline*, meaning that the replica is unavailable because it has failed for reasons such as a crash or a link failure.

Modulo performs $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ on the AEM according to the transition descriptions given in Table 2. For $\mathcal{D}\rightarrow$, Modulo fails zero or more replicas, followed by zero or more write operations from the write sequence (though obviously there should be either non-zero replica failures or non-zero writes). For example, consider the replicaState of a 3-replica system, which can be represented by a tuple $[R_A, R_B, R_C]$ with each element corresponding to a replica's replicaState. Recall that the replicaState is the index of the latest write that the replica has ideally replicated. Thus, a divergence transition that applies a write to replica A, which is replicated to replica B, will change the replicaState from $[1,1,1]$, to $[2,2,1]$. Because replica C did not replicate the write due to a failure, its replicaState does not increase. For $\mathcal{C}\rightarrow$, Modulo returns one or more replicas back to operation and initiates resync. In cases where resync is automatic, the AEM will simply model all online replicas as achieving the latest write index. However, some systems, like Redis, allow manual resync between a subset of replicas, in which case the AEM may explore states with different subsets of replicas resynchronizing. In general, if the type of resync or failure that can occur depends on the abstract AEM state, then the AEM model will specify the type of resync or failure for each $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ in the generated schedules accordingly. If it does not, then the CEM will run the SUT several times with the same schedule, trying out the different SUT-specific failure and resync methods. The CEM, which we discuss in §3.2, specifies many of the details on how replica failures are caused and how Modulo can tell if resync is complete.

Modulo's schedule generator applies symmetry reduction [15] to remove schedules that have identical states. For example, in a system with replicaState of $[3,3,1]$, where replicas A and B have replicated up to write #3 while replica C has only replicated up to write #1, failing replica A or replica B is symmetrical, so Modulo will only produce one schedule for both of those cases. One notable caveat is that some systems, such as ZooKeeper, distinguish one leader replica from the others. We can extend AEMs in a straightforward way by adding a state variable to track which replica is the leader, and a leader and the non-leader will be considered not identical for the purposes of symmetry reduction.

An AEM produces schedules for a CEM to interpret and inject events to a SUT. The following is the schedule we used to find our example ZooKeeper bug, which was generated by using Q/C/Z-DRM (see §3.3) with the modulo parameters $numOps = 5$ and $numReplicas = 3$. The example Figure 2 illustrates what happens between ⑤ and ⑩. Note that we use integer values to indicate the degree for divergence but to identify replicas for convergence (i.e., 0 for A, 1 for B and 2 for C).

$\mathcal{D}\rightarrow$: Divergence, $\mathcal{C}\rightarrow$: Convergence
① $\mathcal{D}\rightarrow$ [0, 0, 1] // introducing divergence by making C commit a write $W(k1, v6)$ while other replicas are failed; then fail C
② $\mathcal{C}\rightarrow$ [0, 1] // introducing convergence by recovering failures of A and B and having them resync; C remains failed
③ $\mathcal{D}\rightarrow$ [0, 1, 0]
④ $\mathcal{C}\rightarrow$ [0, 1] // epoch 4 begins; B becomes a leader
⑤ $\mathcal{D}\rightarrow$ [0, 1, 0] // $W(k3, v8)$ is committed on B
⑥ $\mathcal{C}\rightarrow$ [0, 2] // resync TRUNC; C takes a snapshot
⑦ $\mathcal{D}\rightarrow$ [1, 0, 0] // skipped in the figure
⑧ $\mathcal{C}\rightarrow$ [1, 2] // SNAP resync; truncation fails (Bug)
⑨ $\mathcal{D}\rightarrow$ [0, 0, 1] // $W(k5, v10)$ is committed on C
⑩ $\mathcal{C}\rightarrow$ [1, 2] // resync DIFF; Bug manifests
⑪ $\mathcal{C}\rightarrow$ [0]

3.2 Concrete Execution Model

The purpose of a CEM is to translate the abstract $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ in the schedules generated from an AEM into concrete actions that can be performed on an SUT, to drive it down the individual schedules. For example, a $\mathcal{D}\rightarrow$ may indicate that one of the replicas advances its write index while the others do not, which the CEM may translate as failing 2 replicas and then injecting a write into the SUT. As such, the concrete executor and CEM share some similarities with concrete model-checkers, except that the Modulo's concrete executor only explores $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ sequences specified by the schedules generated by the AEM, and thus only control the aspects of the concrete state that map onto the abstract state

of the AEM, namely whether replicas are online or offline, and what writes are replicated by each replica.

In most cases, $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ can be mapped to a set of SUT-specific APIs to write keys in the SUT and to check if resync has completed. In some cases, CEMs may also need to monitor log files to infer whether the certain aspects of resynchronization, such as leader election, have completed. Finally, in some extreme cases, we may need to instrument the SUT itself to reveal such interfaces to the CEM. For example, if we wanted to make the leader replica explicit when the system does not provide such information, we can replace the default leader election protocol with the one that can explicitly report the result of the leader election to our tool. Different SUTs have different requirements that must be met before they can accept writes. For example, some systems require a leader to be elected before they can ingest writes. These requirements must also be encoded in the CEM so that writes specified in the abstract schedule are correctly applied to the concrete SUT.

For each type of transition specified by the AEM, the CEM may have several ways of realizing that transition, allowing multiple concrete test sequences to be generated from a single abstract schedule. For example, some SUTs treat different types of failures differently (i.e., replica crash vs a network partition). The CEM may run the same schedule but select a different failure type at each $\mathcal{D}\rightarrow$. Similarly, there can be different options during $\mathcal{C}\rightarrow$. Another place where CEM may have several options is what concrete set of writes, in terms of key names and values, will be used to realize the abstract writes in the AEM. In most cases, a set of unique values to a small set of keys suffices.

Below we show how our Q/C/Z-DRM's CEM interprets and injects events into a SUT for realizing divergence and convergence transitions to manifest our example bug. `setData(<k>,<v>)` sets <k> to <v>.

To realize, $\mathcal{D}\rightarrow$ [0, 1, 0]:
① Crash A // no need to crash C, as it is already down
② `setData(<k3>,<v8>)` to B
③ Wait for commit on B
④ Crash B
To realize, $\mathcal{C}\rightarrow$ [0, 2]:
① Restart A
② Restart C
③ Wait for resync completion

3.3 DRM Examples

Overview. Table 3 shows the description of DRMs we have implemented to test ZooKeeper, MongoDB, and Redis in our experiments. We name the DRMs according to the following format: The first letter indicates whether the SUT requires a quorum of replicas (i.e., more than one-half of the replicas must be online) to receive write requests or not. Our DRM models support both systems that require quorum (Q) and those that are stand-alone (S). The second is how replica failures are injected in the model. For failure methods, Modulo can forcibly kill replicas with the signal `SIGKILL` to simulate crash failures (C), suspend replicas with the signal `SIGSTOP` to simulate systems stalled due to a sudden burst of heavy load (S), prevent replicas from communicating to simulate link failures (L) or decommission replicas from a cluster to simulate replica replacement (D). The third is which SUT it is written for, either ZooKeeper (Z), MongoDB (M), or Redis (R). Models are named using the scheme <*quorum_requirement*> /<*failure_modes*>/<*SUT*>.

Also, the user-specified portion of each DRM is presented in Table 3. We had to manually write between 72 to 782 lines of code, which is the result of the effort trying to reduce the manual effort required for each DRM. We put the majority of the code overlapped across DRMs into library or template classes that users can use or extend. We believe we can further reduce the number of codes to write manually. Even for S/CL/R-DRM which required the largest codes for us to write has a large portion of the code that can be further implemented as a library or a template class. Therefore, we think the manual effort required to use DRMs is not heavy and it can become even lighter as the library and templates get mature.

Methodology. In implementing DRMs, we learned a couple of key lessons. First, one should write DRMs in a top-down approach. Think about the most general behavior first. Then, one can extend it by inheriting the most part while overriding only for differences. By doing so, users can reduce the amount of the code they need to write significantly. For instance, write a DRM that injects only one type of failure first. Then, users can write a DRM that can inject multiple types of failures by reusing many lines of code.

Second, focus on the behavior that matters to find target bugs. Users may have a specific type of CFBs they want to find foremost. For instance, ZooKeeper has suffered from errors in transaction log handling. To create more complicated cases, crashing and restarting is important because transaction log is backed by files where a new file is created everytime a new epoch begins. However, focusing on crash failures may not be effective for other systems that rely heavily on full resync using a single transaction log file or that may always use a snapshot resync. For those systems, exploring various network partition failures may be more effective.

Third, pay attention to configuration parameters. Distributed systems rely on various configuration parameter values and their behaviors depend on them. For example, Redis maintains a log of transactions failed replicas missed for a certain timeout period in case a failed replica comes back. Normally, this speeds up resync if the replica returns before the timeout. Modelling divergence behavior to wait longer than the timeout before triggering resync will cause the log to be prematurely discarded and replication fail silently without

attempting to use full resync or SNAP resync.

We now describe some interesting aspects of some models we built in more detail below.

Q/C/Z-DRM. This model is used for ZooKeeper, which requires a quorum of replicas to be online to start servicing requests. This DRM only models crash failures. Since a quorum is required, a $\mathcal{D}\rightarrow$ is only allowed when enough replicas to form a quorum are online, but at the same time $\mathcal{C}\rightarrow$ are not enabled when all replicas are online because there is no divergence to resolve. This model simply uses `kill -9` to send a `SIGKILL` signal, simulating crash failures. Since a quorum is required, the model is restricted in that it must ensure a quorum of replicas is available before it can start injecting write operations, otherwise the write operation will be rejected by the SUT. Also, after $\mathcal{C}\rightarrow$, it needs to pause before executing the next $\mathcal{D}\rightarrow$ in order to ensure that resync is complete. In ZooKeeper prior to version 3.5, log messages are scanned to confirm the existence of a leader that guarantees the resync completion. As of 3.5, it is not the case, so we use timeout.

Q/C/M-DRM. We reuse the same AEM as the Q/C/Z-DRM's. This demonstrates how AEMs can be reused for different SUTs. To confirm the resync completion, we use an MongoDB API call to query the internal document, "`replSetGetStatus`," to retrieve the timestamp of the latest transaction committed on each replica. Then, we wait until those timestamps of every replica becomes same. In case replicas never converge, potentially due to a CFB, it uses an internal time out.

S/S/R-DRM. This DRM does not require a quorum to start servicing clients. It models a chain replication where any replica can become a *sync source* and a *sync target* [2]. Any write committed by a sync source will be replicated to its sync targets. Sync targets cannot have more than one sync source, but sync source can have multiple sync targets. This DRM considers suspend failures instead of crash failures. Also, this DRM starts with an initial state where no replica is connected with another replica. For recovery, we check if the replica is connected with another one and, if not, then we establish new links with other replicas. This model uses `kill -STOP` and `kill -CONT` to suspend and resume Redis processes, respectively. To confirm the completion of resync, this model employs the hybrid of 2 methods. First, it uses the `info` API call to read the `master_link_status`, `master_sync_in_progress`, `aof_rewrite_in_progress` and `rgb_bgsave_in_progress` variables that indicate whether resync is complete. Second, we found that just relying on these variables can cause the CEM to miss some cases when resync is complete. Thus, the CEM also uses a timeout to ensure forward progress. We believe this reduces unnecessarily long timeout, and demonstrates how flexible Modulo can be. Unlike ZooKeeper and MongoDB where resync is automatically triggered by recovering failures, Redis does not automatically trigger resync after a new replica joins. Instead, Redis requires a `slaveof` API call to be explicitly invoked to trigger resync. Thus, this CEM explicitly has replicas establish links between sync sources and a sync targets using the API call, as specified in schedules by its AEM.

S/L/R-DRM. This DRM models link failures only, causing a replica stop replicating data from its sync source. When it recovers a partitioned replica, it considers all possible scenarios of re-establishing replication links. This model simulates link failures using the Redis API command `slaveof no one`, which tells a replica that it has no sync source, causing it to stop replicating data from its sync source. When a link is re-established, the `slaveof` API is used.

S/CL/R-DRM. This model uses both crash and link failures. In addition, $\mathcal{C}\rightarrow$ can pick one of 2 kinds of resync strategies. One is *online resync* and the other one is *offline resync*. Online resync is a built-in resync mechanisms of Redis that gets triggered by re-establishing links between sync sources and sync targets. Offline resync, on the other hand, is a manual resync procedure that can be performed by an administrator. An administrator may manually copy the snapshot of a sync source to a sync target and then may have the sync target start off on the snapshot copy. Because, it considers both types of failures, it generates a larger state-space than the previous AEMs.

3.4 Implementation

Modulo is implemented in Java and comprises roughly 8.4K lines of code. Schedule generation is implemented in about 281 lines of code, and concrete execution takes about 766 lines of code. Our DRMs total 7.3K lines of code where the AEMs and CEMs consist of 2.8K and 4.6K lines of code, respectively, including DRM examples, a library and templates.

A significant part of the DRM implementation consists of SUT log parsing, API interaction and analysis code to infer the state of SUT replicas. To give a concrete example, ZooKeeper records whether a replica is a leader or not in its log message file as follows:

```
LEADING - LEADER ELECTION TOOK - 230
Follower sid: 0 : info : ...
Synchronizing with Follower sid: 0 ...
```

Similarly, a follower replica will log the following messages:

```
FOLLOWING - LEADER ELECTION TOOK - 217
Resolved hostname: 127.0.0.1 to address: ...
Getting a diff from the leader 0x100000009
```

Thus, the DRM can scan log message files of ZooKeeper looking for messages containing either `LEADING` or `FOLLOWING`

[2] See: https://redislabs.com/ebook/part-2-core-concepts/chapter-4-keeping-data-safe-and-ensuring-performance/4-2-replication/4-2-3-masterslave-chains/

Name (Parameters)	**AEM**	**CEM**	**Lines of Code** (AEM/CEM/Total)
Q/C/Z-DRM ($numOps = \{1..5\}$, $numReplicas = \{3..5\}$)	Only considers crash failures. $\mathcal{C}\rightarrow$ ensures the quorum exists before $\mathcal{D}\rightarrow$. Crashes all replicas at the end of $\mathcal{D}\rightarrow$	Using kill -9 to send SIGKILL for crash failures. Confirm the quorum exists before writes. Using log scanning to confirm the leader for versions before 3.5, but, as of 3.5, relying on timeout.	USER 54/59/113 LIB 339/620/959
Q/C/M-DRM (same as above)	Same as Q/C/Z-DRM	Using an API to retrieve "replSetGetStatus" and compare timestamps of the last transaction on each replica to wait for resync completion	USER 54/117/171 LIB 339/907/1246
S/S/R-DRM ($numOps = \{1..2\}$, $numReplicas = \{4\}$)	Only considers suspend failures. Considers all replicas initially partitioned. As recovering suspend failures, establish links between the recovered replica and an online replica.	Using kill -STOP and kill -CONT to simulate suspend and resume. Using the 'info' API and timeout to wait for resync completion. Using the 'slaveof' API to trigger resync	USER 33/39/72 LIB 955/1240/2195
S/L/R-DRM ($numOps = \{1\}$, $numReplicas = \{3\}$)	Only considers link failures between an arbitrary pair of replicas. Considers replicas initially connected in a single 'slave chain.'	The 'slaveof' API is used for link failures and recoveries. Initially, forming links as a single slave chain.	USER 0/110/110 LIB 955/1240/2195
S/CL/R-DRM ($numOps = \{1,2\}$, $numReplicas = \{2\}$)	Considers both link and crash failures. Considers two types of resync strategies: online resync and offline resync.	For the offline resync strategy, a script copying over snapshots and starting up a replica with the snapshot is used.	USER 405/377/782 LIB 955/1240/2195

Table 3: The summary/comparison of DRM Examples. The naming convention indicates the quorum requirement, failure modes, and the target SUT. Below the name, we also present modulo parameter values we used. AEM and CEM give more detailed descriptions of each subcomponent of the given DRM example. Only the differences of each DRM compared to the one above is specified. The rightmost column shows the lines of code (LOC) for user-specified portion of DRMs (USER) and for a library and templates users use (LIB)—only the part directly interfacing with user-specified portion is counted. Since each DRM may use different template, there can be difference in LOC for LIB.

indicating which role they have switched to. The DRM needs to keep track which replica is the leader in order to successfully inject write operations to cause divergence, because only the leader replica can ingest write operations.

4 Evaluation

We present our results from running Modulo on 3 mature, open-source, distributed storage systems: ZooKeeper, MongoDB, and Redis.

4.1 Bug Discovery

In Table 4, we summarize the CFBs found and tabulate the time Modulo took to find each of the bugs, as well as the number of transitions of the schedule manifesting the bug. More detailed description is provided in Appendix A. Those bugs labelled with "**New!**" had not been reported until we discovered. Our evaluation study has been conducted between 2017 and 2020— ZooKeeper Bug #1 and ZooKeeper Bug #2 were found in 2017, while ZooKeeper Bug #3, ZooKeeper Bug #4, and ZooKeeper Bug #5 were discovered in 2020. We note that some bugs are quite complex, requiring a specific sequence of more than 10 transitions to trigger the bug. Also, DRM state space is evaluated in terms of the number of schedules generated for different DRMs and different modulo parameter values. Lastly, we show the state coverage measurement over time for the setting used to test the most recent ZooKeeper version.

4.2 Testing Performance

Table 4 also shows how much time our prototype took to find each bug mentioned above. The performance was measured on a machine with an 2.83GHz Intel Core2 Quad CPU and 8GB of RAM. We found that the limiting factor for the testing speed of Modulo is (1) the speed of the underlying distributed system and (2) how easily Modulo can infer that the system has converged after a $\mathcal{C}\rightarrow$ so that it can inject a $\mathcal{D}\rightarrow$. In general, distributed systems do not prioritize speed during replication operations, and the lack of interfaces to infer when they are done can lead to slower testing as in the case of MongoDB. Without an interface that allows the CEM to tell if convergence had been achieved, Modulo had to check the timestamp of the keys on every replica to see if they were the same. In addition, MongoDB can take a long time to achieve convergence, forcing the CEM to use a very high timeout value (10 minutes in our experiments). In contrast, the CEMs for ZooKeeper and Redis can infer that convergence has occurred in other ways that do not require either checking or a timeout. We believe it should be possible to modify MongoDB to allow Modulo to infer whether convergence has

Bug ID	DRM	Root Cause	Elapsed Time	Time/ Schedule	# of Trans.
ZooKeeper Bug #1(**New!**) [8]	Q/C/Z-DRM	Fail to truncate operations due to missing invocation	11 hours	33 sec	11
ZooKeeper Bug #2(**New!**) [9]	Q/C/Z-DRM	Fail to truncate operations due to file handling mistake	2 hours	39 sec	11
ZooKeeper Bug #3(**New!**) [10]	Q/C/Z-DRM	Fail to replicate operations due to an incomplete log	23 min	33 sec	7
ZooKeeper Bug #4(**New!**) [11]	Q/C/Z-DRM	Fail to truncate operations due to a pointer handling mistake	47 min	30 sec	10
ZooKeeper Bug #5(**New!**) [12]	Q/C/Z-DRM	Fail to truncate operations due to missing invocation	20 hours	37 sec	10
MongoDB Bug #1	Q/C/M-DRM	Fail to truncate operations due to incomplete timestamp information	18 min	6 min	3
MongoDB Bug #2(**New!**) [36]	Q/C/M-DRM	Fail to replicate operations due to incomplete protocol design	4 hours	5 min	5
Redis Bug #1 [25]	S/S/R-DRM	Fail to invoke snapshot sync due to incomplete protocol design	6 hours	6 min	6
Redis Bug #2 [53]	S/CL/R-DRM	Fail to replicate operations due to lacking resync related information	11 min	14 sec	4
Redis Bug #3 [53]	S/CL/R-DRM	Fail to replicate operations due to lacking resync related information	2 min	6 sec	3
Redis Bug #4 [22]	S/L/R-DRM	Fail to truncate operations due to incomplete protocol design	2 min	33 sec	2

Table 4: CFB Analysis Summary.

DRM	*numOps*	*numReplicas*	# of Schedules
Q/C/Z	1	3	6
	2	3	80
	3	3	1035
	4	3	13381
	5	3	172993
	3	4	3428
	3	5	54655
S/S/R	2	4	13586
S/L/R	2	3	263
S/CL/R	1	2	8
	2	2	96

Table 5: DRM State Space Size (# of Schedules).

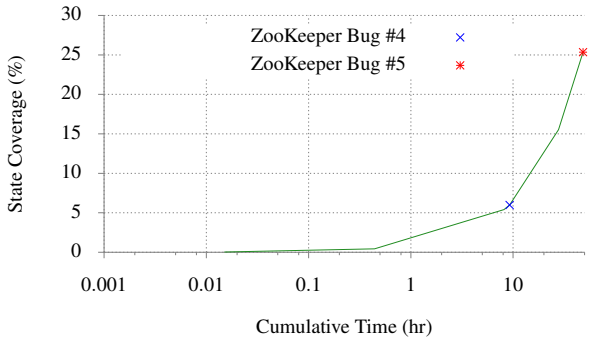

Figure 4: DRM State Coverage graph. We show the state coverage measurements from testing ZooKeeper 3.7. X-axis shows the cumulative time and Y-axis shows the state coverage ratio based on the number of schedules executed. Also, we mark when we found ZooKeeper Bug #4 and ZooKeeper Bug #5.

occurred without the needing to rely on timeouts and intend to explore this in the future.

4.3 DRM State Exploration

Table 5 shows several examples of the DRM state space size in terms of the number of schedules generated. As the number of *numOps* or *numReplicas* increase, the number of schedules generated quickly grows. The largest number of schedules were generated for Q/C/Z-DRM with $numOps = 5$ and $numReplicas = 3$ which we used to test ZooKeeper 3.4.11 and found the example bug. Nevertheless, as we reason in §5, the state space for Modulo to search is much smaller than explicit state model-checkers.

In Figure 4, we measured how state space coverage is increased during our evaluation finding bugs in the version of ZooKeeper, 3.7. We fixed *numReplicas* at 3 and varied *numOps* from 1 to 4, which increases the time taken. For each *numOps* setting, we ran tests separately one after another. Bugs can be found by schedules with no particular probability distribution. Considering this, it is important to run each test exhaustively not to miss any bug. Our design choice to split AEM and schedule generation from the concrete test execution by CEM also enables easy parallelization. Indeed, we when parallelize our tests, we get a linear increase in throughput.

5 Discussion

Modulo is designed to detect CFBs efficiently by only exploring system states that are relevant to its DRMs, which only model states and events relevant to $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ state transitions. In contrast to traditional system model-checking [27,29,35,41,42,44,52,61], Modulo is less complete, meaning it may miss bugs that manifest due to events like lock acquisition and thread interruption, as these are not captured by $\mathcal{C}\rightarrow$ and $\mathcal{D}\rightarrow$ in DRMs. However, by abstracting away states and events not relevant to DRMs, Modulo is able to explore considerably deeper bugs that, in some cases, take 10 or more transitions to manifest. Furthermore, Modulo finds

these bugs in real, concrete systems rather than in abstract models, making reproduction and confirmation of bugs much simpler.

Compared to distributed systems random testing, such as that employed by Jepsen [38], Modulo is more systematic, complete and exhaustive. Random testing can find many crucial bugs by randomly injecting external events; however, it is neither systematic, complete nor exhaustive for the following reasons. First, it does not model how the underlying distributed system works, including key concepts such as divergence and convergence. Hence, random testing may miss corner cases regarding divergence and convergence. Second, fuzzing does not control nondeterministic events. For instance, without delaying the timing of a crash-failure injection, a replica may crash before ensuring a write injected before the crash is committed. Third, random testing cannot reproduce bugs the way Modulo can, making the analysis and reproduction of bugs more challenging.

Modulo significantly reduces the severity of state explosion by taking a simple abstract model and mapping it onto transitions on a real concrete system. This allows Modulo to find deeper bugs than systems that attempt to explore more complex state spaces. To illustrate, Modulo's AEM's typically model 3-replica systems with between 1 and 5 writes. Before each write in the sequence, a replica can typically (1) do nothing, (2) receive or replicate the write operation, or (3) failure/recovery (e.g. crash/restart). Since replicas may have an arbitrary number of transitions where they may do nothing, we typically cap the maximum number of transitions in a replica an AEM may have at around 8. Thus we can estimate a rough upper bound for the state space of such a system as $3^{(8\times3)}$ or roughly 300 million. In practice, the number of reachable states is far less, because replicas can accept writes only when they are online, and other AEM-specific restrictions such as quorum requirements further limit the transitions that replicas may execute. In practice, Modulo's targeted approach leads to have AEMs produce anywhere from few schedules to tens of thousands of schedules, which is about 10 to 10^5 of states. In comparison, models checked by explicit state model checkers may contain more than 10^{20} or even 10^{250} states [17]. Our relatively small DRMs mean that Modulo is able to explore more scenarios than traditional model checkers could in given time, increasing the likelihood of finding target bugs, without exploring states that are irrelevant to find those specific types of bugs.

Meanwhile, we also note that a targeted approach can be seen as a disadvantage for finding many of various types of bugs. Indeed, as Modulo is targeted to find a specific type of CFBs, the number of bugs we found from our evaluation is relatively low. Nevertheless, we envision that the number of bugs found by Modulo can be increased, as developers who are more expert of each SUT can develop a larger collection of DRMs that are more effective to explore corner cases.

We acknowledge that Modulo does depend on domain knowledge of the system-under-test to specify DRMs, and this would also be required to apply Modulo to other types of distributed systems. In our experience, a single user without previous experience may conservatively take about 2 weeks to learn about the system-under-test and 2 weeks to write the first DRM. When we first applied Modulo to Redis, using the same model as ZooKeeper and MongoDB did not lead to bugs, because Redis, started as an in-memory key-value store, has a simpler persistent storage mechanism. Subsequently, we found exploring suspend or link failures enabled Modulo to trigger more complex functionality. Modulo's methodology, abstraction and concrete execution, is not necessarily restricted to key-value stores but can be applied to other distributed systems. Paxos-based systems can be effectively tested using Modulo by modelling message/thread interleaving alongside failure injections.

Our main limitation is that the schedules generated by the abstract model must be run on the real SUT, which executes much slower than an abstract model would. In addition, many operations require pauses and timeouts before they can complete. For example, a system may not consider a replica failed until a certain time has passed, and leader election must complete after replicas recover before writes can be ingested by the system. Since they are not always possible to avoid, in practice we find that these timeouts are the ultimate limit on how fast Modulo can find bugs. We think virtualizing clocks and fast-forwarding time may help [42].

6 Related Work

Distributed Systems Testing. Some previous proposals for distributed systems testing employ state-space exploration. However, their state-spaces are more focused on interleaving concurrent internal events, such as thread scheduling or network message delivery [20, 28]. Also, previous work presents a tool for injecting network-partitioning failures for cloud systems [7], yet it does not inject crash failures, and it requires an OpenFlow-capable hardware component to simulate network-partitioning. Jepsen is an open-source testing tool that injects various types of failures into distributed database systems [38]. However, unlike Modulo, Jepsen randomly generates inputs, which implies that the state space for input sequences cannot be efficiently reduced without missing corner cases and, in some cases, reproducing bugs may be very difficult due to the nondeterministic ordering of events that Jepsen does not control. Furthermore, Jepsen does not record any information about the state exploration, therefore it cannot provide any guarantee about the state-space coverage. There has been a work that focuses on interleaving low-level file system operations across distributed replicas along with crash failures [6], but it is limited to crash injection only and does not test the divergence and convergence behaviors of distributed systems.

Model-based testing systematically derives test cases from

an abstract model of the SUT [14, 37, 45, 48, 49, 54]. Dalal et al. [18] devised a method that can generate various input parameter values for tests from the abstract model called the Test Data Model. A technique that can derive test cases for system testing from an UML statechart was developed by Offutt et al. [47]. Also, Gargantini et al. [24] propose to use model checking to derive test cases from an abstract model. In addition, Andrews et al. [13] came up with a technique that models a web application as a finite state machine to generate tests. Also, Yang et al. [63] applied model-based testing to find security flaws in about 500 implementations of OAuth 2.0. More recently, Davis et al. [19] used model-based testing to ensure specification-implementation conformance of distributed systems. However, previous proposals for model-based testing do not look for CFBs. Also, none of the existing model-based studies test the divergence and convergence of replicated distributed storage systems as the SUT.

Distributed Systems Model Checking. Model checking has been extensively studied and used to prove the correctness of complex systems and to find bugs in them. Clarke and Emerson [16] were the first to propose model checking, which exhaustively explores the state space of abstract models specified in temporal logic. Dill developed a model checker called Murphi [21], which is used to prove the correctness of various systems, including distributed shared memory systems. SPIN is another popular abstract model-checking tool [30]. More recently, Lamport [40] developed TLA+, a specification language, and TLC, a model checker for TLA+, which has been used by Amazon, showing the practicality of model checking in the industry. Nevertheless, abstract model-checking cannot find bugs in implementations directly.

Concrete model-checking is used to employ an implementation as a model to explore directly. Musuvathi et al. [44] proposed a concrete model-checker using implementations as the model to verify. Godefroid [26] also proposed the same idea around the same time. Killian et al. [35] devised a technique that enables checking for not only safety properties but also liveness properties. Lin et al. [42] developed a black-box concrete model-checker that does not need to know about the source code by interposing events at the interface layer between the target system and the underlying operating system. Model checking not only proves the correctness of the system but also predicts if the implementation execution is driving the system to faulty states, steering the system execution away to prevent this [61]. Simsa et al. [52] explored the generalization of concrete model-checking to provide the flexibility for determining the level of controls for nondeterminism. Recently, concrete model-checkers have been improved to detect deep bugs by exploring scenarios involving multiple failures [41, 43]. Unlike Modulo, previous concrete model-checkers explore various tightly controlled sequences of concurrent internal events.

7 Conclusion

Modulo mitigates the traditional state-explosion problems of systematic model-checking approaches to find CFBs by abstracting away all states and state transitions that are not related to the concepts of convergence and divergence. Modulo applies schedules derived from state explorations of small abstract models of such systems to real distributed systems. Our work identified several factors that lead to such bugs, which include (1) employing several resync or failure-handling mechanisms whose interactions are difficult to foresee, (2) hard limits or inadequate designs for handling large amounts of divergence, and (3) assumptions about length of time that replicas may have failed and failures that span events like leader transitions. While it is beneficial to generate counter-examples that trigger the bugs on real systems, we find that this also slows down the speed at which Modulo can find bugs, as it must examine the state of the system at to determine if it can inject the next input or not, and it must wait for the distributed system itself to complete its replication operations before adding divergence.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Nathan Bronson, for their wonderful guidance and feedback. We also thank Ashvin Goel and Eyal de Lara for their helpful comments. This research was supported by NSERC Discovery Grant RGPIN-2018-05931, a Canada Research Chair, and NSF CNS-1749711.

References

[1] Apache HBase. https://hbase.apache.org.

[2] Couchbase Server. https://www.couchbase.com.

[3] MemcacheDB. http://memcachedb.org.

[4] MongoDB. https://www.mongodb.com/.

[5] Riak. http://basho.com/products.

[6] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 151–167, Berkeley, CA, USA, 2016. USENIX Association.

[7] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of*

the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, pages 51–68, Berkeley, CA, USA, 2018. USENIX Association.

[8] anaud. Synchronization code on the follower does not properly truncate uncommitted write resulting in data inconsistency, 2017. https://issues.apache.org/jira/browse/ZOOKEEPER-2945.

[9] anaud. The truncate() function in filetxnlog.java may fail to properly remove an uncommitted write resulting in data inconsistency, 2017. https://issues.apache.org/jira/browse/ZOOKEEPER-2946.

[10] anaud. Convergence fail when a follower tries to resync with a leader having incomplete commitlog, 2020. https://issues.apache.org/jira/browse/ZOOKEEPER-3972.

[11] anaud. Convergence fails when a follower missed the committedlog synching with the leader if it was an old leader and the leader falls back to send snapshot., 2020. https://issues.apache.org/jira/browse/ZOOKEEPER-3946.

[12] anaud. truncate in filetxnlog.java is buggy and fails to correctly truncate a file containing a single transaction only the follower saw, 2020. https://issues.apache.org/jira/browse/ZOOKEEPER-3947.

[13] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with FSMs. *Softw. Syst. Model.*, 4(3):326–345, July 2005.

[14] Lionel C. Briand and Yvan Labiche. A UML-based approach to system testing. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, UML '01, pages 194–208, London, UK, UK, 2001. Springer-Verlag.

[15] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, pages 147–158, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[16] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[17] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, page 176–194, Berlin, Heidelberg, 2001. Springer-Verlag.

[18] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[19] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. Extreme modelling in practice. *Proceedings of the VLDB Endowment*, 13(9):1346–1358, May 2020.

[20] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 249–262, Berkeley, CA, USA, 2016. USENIX Association.

[21] David L. Dill. The Murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 390–393, London, UK, UK, 1996. Springer-Verlag.

[22] fdingiit. Questions about potential data inconsistency / unexpected FSYNC base on 4.0.2, 2017. https://github.com/antirez/redis/issues/4407.

[23] Camille Fournier. leader/follower coherence issue when follower is receiving a diff, 2010. https://issues.apache.org/jira/browse/ZOOKEEPER-962.

[24] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, October 1999.

[25] GeorgeBJ. Replication inconsistent issue, 2015. https://github.com/antirez/redis/issues/2694.

[26] Patrice Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[27] Rachid Guerraoui and Maysam Yabandeh. Model checking a networked system without the network. In *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*, page 225, 2011.

[28] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud

recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 238–252, Berkeley, CA, USA, 2011. USENIX Association.

[29] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 265–278. ACM, 2011.

[30] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[31] Hailin Hu. Inconsistent query results between primary and secondary, 2017. https://jira.mongodb.org/browse/SERVER-31663.

[32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[33] Jacky007. Data inconsistency when follower is receiving a diff with a dirty snapshot, 2012. https://issues.apache.org/jira/browse/ZOOKEEPER-1549.

[34] Vishal Kathuria. Data inconsistency when the node(s) with the highest zxid is not present at the time of leader election, 2011. https://issues.apache.org/jira/browse/ZOOKEEPER-1154.

[35] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of NSDI'07: 4th USENIX Symposium on Networked Systems Design and Implementation*. NSDI, 2007.

[36] Beom Heyn Kim. Initial sync not replicating old oplog entries may have a stale node give up resync and permanently stay in recovering state, 2018. https://jira.mongodb.org/browse/SERVER-35774.

[37] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from uml state diagrams. *IEE Proceedings - Software*, 146(4):187–192, Aug 1999.

[38] Kyle Kingsbury. Distributed systems safety research. https://jepsen.io/.

[39] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[40] Leslie Lamport. Specifying concurrent systems with TLA+, 1999.

[41] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, 2014.

[42] Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of NSDI'09: 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI, 2009.

[43] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 20:1–20:16, New York, NY, USA, 2019. ACM.

[44] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI):75–88, 2002.

[45] Clementine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jezequel. Automatic test generation: A use case driven approach. *IEEE Trans. Softw. Eng.*, 32(3):140–155, March 2006.

[46] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.

[47] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, UML'99, pages 416–429, Berlin, Heidelberg, 1999. Springer-Verlag.

[48] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Softw. Test., Verif. Reliab.*, 13(1):25–53, 2003.

[49] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 105–118, New York, NY, USA, 1992. ACM.

[50] Salvatore Sanfilippo. Redis. https://redis.io/.

[51] Andy Schwerin. Multi-updates may fail to detect replica set primary step-down, leading to inconsistency., 2014. https://jira.mongodb.org/browse/SERVER-12516.

[52] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic evaluation of distributed systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, page 3, USA, 2010. USENIX Association.

[53] soloestoy. Redis 4.x PSYNC2 & RDB: data inconsistency between master and slave, bug located, 2017. https://github.com/antirez/redis/issues/4316.

[54] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22(11):777–793, November 1996.

[55] Jeremy Stribling. Missing data after restarting+expanding a cluster, 2011. https://issues.apache.org/jira/browse/ZOOKEEPER-1319.

[56] Jeremy Stribling. Data inconsistencies and unexpired ephemeral nodes after cluster restart, 2012. https://issues.apache.org/jira/browse/ZOOKEEPER-1367.

[57] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project Voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[58] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *The 24rd ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[59] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[60] Ryan Witt. Secondary keeps getting into an inconsistent state, 2014. https://jira.mongodb.org/browse/SERVER-13222.

[61] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, volume 9, pages 229–244, 2009.

[62] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 157–168, New York, NY, USA, 2014. ACM.

[63] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on OAuth 2.0 implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 651–662, New York, NY, USA, 2016. ACM.

[64] Christian Ziech. Data loss after truncate on transaction log, 2012. https://issues.apache.org/jira/browse/ZOOKEEPER-1489.

A Bug Description

A.1 ZooKeeper Bugs

We discovered 5 new bugs in ZooKeeper by finding 2 new bugs in version 3.4.11, 1 new bug in version 3.5.8 and 2 new bugs in version 3.7.0. We reported them to the ZooKeeper developers and reported bugs were designated as ZooKeeper Bug #1, ZooKeeper Bug #2, ZooKeeper Bug #3, ZooKeeper Bug #4 and ZooKeeper Bug #5 [8–12]. We got confirmation for ZooKeeper Bug #1 and ZooKeeper Bug #3. ZooKeeper Bug #1 is described in §2.3. All our experiments used 3 replicas. Initially, all 3 replicas are online and synchronized with a initial key-value set. Also, the Q/C/Z-DRM implementation for ZooKeeper crashes all replicas after each $\mathcal{D} \rightarrow$ and implements $\mathcal{C} \rightarrow$ by restarting replicas in the quorum, which automatically triggers resync between each replica and the elected leader.

As described earlier, ZooKeeper implements 2 different resync mechanisms: DIFF resync and SNAP resync. Conflict-resolution logic often fails to correctly truncate transaction logs when SNAP resync is used, which results in persistent inconsistency. This problem is further exacerbated by an apparent reluctance on the part of the developers to truncate transaction logs, perhaps out of a conservative preference not to lose data unless absolutely necessary, which results in cases where logs should have been truncated but were not. Other complexities, such as using multiple log files instead of a single log file and incorrect assumptions about those files, contributed to other bugs. We saw similar bugs are recurring and stem from the similar portion of the resync implementation. It shows that the complexity of the resync mechanisms in ZooKeeper has been the major source of CFBs.

A.2 MongoDB Bugs

We run Modulo on MongoDB version 3.0.0 and discovered 2 bugs. One was fixed in later versions of MongoDB by upgrading their replication protocol, Replica Set Protocol version), from Protocol Version 0 to Protocol Version 1. The

other one was new and we reported it on the MongoDB Bug Database [36]. We used 3 replicas in the DRM and the same initial state as we did with ZooKeeper.

The first MongoDB bug we discussed occurred because the developers failed to anticipate the situation where a primary commits some operations that have not been replicated to other replicas, operations only the primary is thus aware of. Modulo found the bug and the bug had never previously reported. However, the bug does not manifest on the latest version of MongoDB. On further inspection, we found that Replica Set Protocol Version 0, which had this bug, was replaced by Replica Set Protocol Version 1, which was implemented in MongoDB version 3.2 and became the default protocol after version 3.6. Thus, this particular bug was not fixed directly by developers, but instead eliminated when the afflicted protocol was replaced by a completely new protocol. With Modulo's bug report, we estimate that the bug could have been fixed by changing 10's of lines of code, but instead was fixed when the entire protocol was re-implemented, which required changing 12 files containing roughly 7.4K lines of code.

The second bug demonstrates the perils of simultaneously using several resync mechanisms. Having several resync mechanisms allows MongoDB to select among them to improve efficiency, but the mechanisms have slightly different side effects, which, under the right circumstances, can combine to put the system into an unrecoverable state.

A.3 Redis Bugs

Modulo found 4 CFBs [22, 25, 53] in Redis versions 2.8.0 and 4.0.0. Upon further examination, we found all had been previously reported.

In Redis, crash failures always lead to the SNAP resync, which simply replicates the entire state of the sync source to the sync target. As mentioned earlier, this trivially guarantees convergence since the sync target is now a mirror of the sync source. To trigger more complex resync mechanisms, Modulo required DRMs that could exercise other replication and failure recovery mechanisms that Redis provides. Unlike Zookeeper and MongoDB, this required the development of 3 different DRMs, and each DRM was responsible for finding at least 1 CFB. We found it useful to have separate DRMs as each DRM could separately exercise some of Redis' features, while combining them would have resulted in a larger state space and more schedules to explore.

In terms of time and effort, Redis took the most: a novice Redis user took a couple of weeks to initially write each DRM, where most of the time was spent understanding Redis' mechanisms and API. The Redis DRMs are also roughly 2-3 times larger and more complex than ZooKeeper and MongoDB DRMs because replicas may specify any sync source to resync from, giving more possibilities. Qualitatively, we feel the effort to construct DRMs is similar to that of writing a specification in a formal specification language such as TLA+, which has been cited as an acceptable cost by developers at Amazon [46]. The user needs to understand the important properties of the system they want to test, and be able to abstract them away from implementation details. In addition, the user must also be able to reduce a system to a smaller number of replicas and smaller number of keys to reduce the state space. However, unlike model checkers such as TLA+, which run on an abstract state machine representation of the SUT, Modulo marries the advantages of a reduced state space produced by the manual abstraction, with the ability to reproduce the bugs found using a counter-example of real inputs that can be run on the concrete system.