Formal Operations Specifications for Reliable System Management

Jinlang Wang
University of Wisconsin-Madison

Jiawei Tyler Gu
Databricks

Ruize Tang
Microsoft Research Asia

Tej Chajed
University of Wisconsin-Madison

Tianyin Xu
University of Illinois Urbana-Champaign

Xudong Sun
University of Toronto

Abstract

Large-scale production systems are largely operated by management software like Kubernetes operators and increasingly by AI agents. Production systems have diverse functionalities, interfaces, and fault-tolerance properties; hence, their management semantics are complex and less standard. Today, management-plane software is largely implemented based on experience and documents of the managed systems, where correct management is a best effort.

We advocate for the practice of writing *formal operations specification* (FOS) as a formal contract to precisely describe expected behaviors of the production system and to enable systematic checking and verification of management-plane software and agents. We argue that the FOS should be part of standard system interface to enable reliable system management. We suggest that every production system include an FOS and share our experience of writing FOSes for real-world systems; we show that writing and maintaining an FOS is affordable and can potentially be automated.

CCS Concepts

• **Computer systems organization** → **Reliability**; • **Software and its engineering** → **Formal software verification**.

Keywords

Formal Specifications, Reliability, Kubernetes Operators, Model checking

ACM Reference Format:
Jinlang Wang, Jiawei Tyler Gu, Ruize Tang, Tej Chajed, Tianyin Xu, and Xudong Sun. 2026. Formal Operations Specifications for Reliable System Management. In *The 25th ACM Workshop on Hot Topics in Networks (HotNets '26), November 16–17, 2026, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3847290.3859962

This work is licensed under a Creative Commons Attribution 4.0 International License.
HotNets '26, Salt Lake City, UT, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3029-0/2026/11
https://doi.org/10.1145/3847290.3859962

1 Introduction

Large-scale production systems have switched from human-centric operation to management software (e.g., Kubernetes operators [12, 17]) and emerging AI agents [11, 24, 38, 43]. While autonomous management planes eliminate human errors, their own correctness has become a major concern for production system reliability in recent years—buggy management software caused widespread failures [19, 20, 40, 44].

The key challenge of management-plane correctness is to understand and respect the semantics of the managed systems, e.g., preconditions of management operations. Unfortunately, software systems have much more diverse functionalities, interfaces, and fault-tolerance properties; as a result, their management semantics are more complex and less standard. For example, different SQL databases can have drastically different management interfaces and may require different operations for the same management task.

Today, the management planes of production systems are mostly implemented based on experience and documents, where correctness is a best effort. However, we find management interfaces are often not well documented but in fact have implicit, subtle semantics. Given that management code is often not written by the developers of the managed system (let alone proprietary systems), ensuring management-plane correctness becomes particularly difficult.

Figure 1 shows a real-world example to illustrate the aforementioned problem. In this example, PostgreSQL[1] exposes a management interface, `failover`; the interface is used by management software, in this case a Kubernetes operator [45], for automatically switching the primary role to a backup. However, correct `failover` operations are non-trivial, because they must satisfy at least three preconditions: (a) the target backup must be in a `ready` state, (b) the backup must

[1]PostgreSQL uses external services like Patroni for distributed high availability (HA); in this paper, we do not differentiate official PostgreSQL and the Patroni service; the running system is referred to as a PostgreSQL cluster.

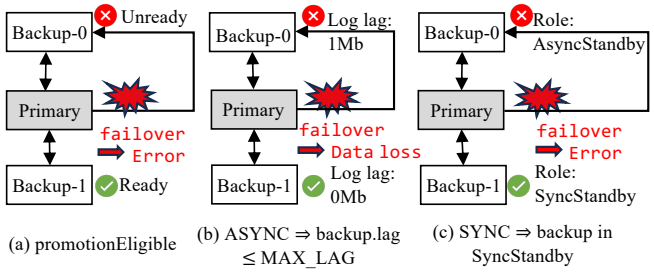

Figure 1: Three incorrect management operations issued by a PostgreSQL operator that led to system failures. Each failure was caused by violating a different precondition required by PostgreSQL's failover operation.

be caught up with the primary in asynchronous replication mode, and (c) the backup must be a sync standby in synchronous replication mode. Violating any of these preconditions leaves the PostgreSQL cluster unavailable. Unfortunately, all three preconditions were discovered by the operator developers only *after* the failures—none of them were documented in the user manual at the time.

Existing methods do not close this semantic gap. Enhancing documentation may help, but documentation is often incomplete, ambiguous, and disconnected from testing and analysis pipelines. A study of operator failures in the field found that erroneous interactions with the managed application are the single largest cause, accounting for 42% of the failures studied [20]. Recent work built automated testing and analysis tools [19, 40] for operator-like programs. However, they target bugs residing in the operators, and cannot find interface bugs. Recent efforts on verifying Kubernetes controllers [7, 41] do not address this problem either, as they do not reason about application semantics. The goal of this work is to advocate for the practice of writing *formal operations specifications* (FOS)[2] as a formal contract with management-plane software or agents. An FOS could precisely describe expected behaviors of the system under management and enable formal reasoning of management software and agents (especially in complex scenarios with failures and unexpected events). We argue that an FOS should be an essential part of the system interface to enable reliable production management.

In this paper, we share our experience of writing FOSes for real-world systems such as PostgreSQL. In fact, there are many ways to write a specification for a given system, and it was initially unclear which was the right choice for operations specifications. Our three criteria were (1) to serve as a precise document of the management interface, (2) to prevent real bugs, and (3) to be easy to write and maintain for regular developers (not formal-methods experts). We soon realized that typical API specifications like OpenAPI Specification for REST APIs [31] that list the constraints of API parameters and their error codes cannot help express the correctness properties, such as those described in Figure 1. Instead, the FOS should be expressed in the form of a system model and its invariants. Any valid management operations should satisfy the invariants and trigger state transitions.

We therefore write the FOS using the P specification language [15]. We chose P because it is a state-machine-based language for formally modeling and specifying distributed systems; it is arguably more friendly to developers compared with other specification languages like TLA$^+$ and Dafny. Specifically, we use P to write formal models of the managed system, the operator, and the Kubernetes API, as well as their invariants. A P model describes each component as a *machine* that has a set of *states* that react to incoming events, update local variables, and *send* events to other machines. We can check the specifications statically through model checking or dynamically through runtime monitoring.

We find that the key challenge of writing the FOS is to determine the level of abstraction—an FOS should describe the system from a management-plane perspective, instead of exposing unnecessary details of the system implementation. We share our practice of writing the FOS for PostgreSQL, which is generally applicable to multiple systems we have tried, including CockroachDB, Solr, Kafka, and RabbitMQ. Our experience shows that writing and maintaining an FOS is affordable and can potentially be automated. We also show that the FOSes are useful—we use them to check four Kubernetes operators and find 39 new bugs.

2 Specification styles

To check or verify the correctness of management software, we need to formalize the specification of what it means for correct management of a target system, or how the target system wants to be managed. We realize that none of the common styles of formal specifications is a good fit for FOS.

First, API specifications that describe the constraints of input parameters (e.g., OpenAPI Specification for REST APIs) are insufficient. For example, in Figure 1's example, all the erroneous management operations in fact issue valid API calls to failover with valid parameter values.

We realize that it is important to model the *states* of the system under management; a management operation essentially changes system states. The FOS defines the invariants on the system states, which can be encoded as the pre- and post-conditions of specific management operations. In other words, we expect FOS to include (1) a formal model of the

[2]We borrow the term *operations specifications* (aka OpSpecs) in aviation administration, which are regulatory documents that dictate approved routes, aircraft, flight rules, and safety requirements of an aviation operator.

system under management as a state machine (or more generally a transition system), and (2) invariants that describe correctness properties of the system under management.

Moreover, the system model must describe the system implementation, not the high-level protocol (e.g., primary-backup replication), as FOS targets production systems, not the protocol—implementations of the same protocol can have very different management interfaces and semantics. In our experience, the most effective way to write FOS is to follow the source code of the target system, not its documentation (which is often incomplete, outdated, and ambiguous).

While prior work has presented ways to write formal system models, e.g., in TLA$^+$ [32, 42], Verus [41], Alloy [37], and programming languages [5, 22, 23], *we find those models cannot serve as FOSes, because they are describing the internal behaviors of the target system (to verify its implementation), instead of concerning how the system is managed.* Take Figure 1 as an example: the FOS does not concern whether the primary-backup protocol of PostgreSQL is implemented correctly; instead, it concerns whether the management software (e.g., any Kubernetes operators) satisfies the invariants defined by PostgreSQL when issuing management operations like failover. Formal models and specifications designed for Kubernetes controllers [1, 7, 26, 41] (Kubernetes operators are one kind of controller) do not apply either—*an FOS is a formal contract between the system and its management software, not a general specification for controller code.*

We ended up using P [15] to write FOSes in part because we were looking for a specification language that is friendly to regular developers (so they are more willing to write FOSes). But, as formal specifications are increasingly generated by AI [9, 14, 28], the choice of language may no longer make a major difference. FOS is not specific to P.

3 Formal operations specifications

The key challenge of writing an FOS is to decide the level of abstraction. In principle, an FOS should describe states that are important to the management plane, while abstracting away states or behaviors that are internal to the system. The challenge is to draw the boundary at what the system must expose for the FOS to be checkable.

In this section, we describe our recipe for writing FOSes using PostgreSQL and its Kubernetes operator in Figure 1 as a running example. Our recipe has three steps. First, we specify each management interface as a Hoare triple: starting from the operator's call into the managed system, we derive its precondition from the conditions the system checks on a request, and its postcondition from the management state it establishes on success (§3.1). We then construct a P model of the system, including only the state its precondition and postcondition reference or the operator observes, and abstracting the rest away; the interface's behavior is modeled as actions over this state. We also define invariant monitors that encode the precondition and postcondition as assertions (§3.2). Finally, we compose the FOS with formal models of the operator and Kubernetes (also written in P) and model-check their composition; the models are validated against real execution traces to eliminate model-code gaps (§3.3).

3.1 Specifying management interfaces

A system exposes management interfaces for querying or updating its management state. For example, in Figure 1, failover is a management interface exposed by PostgreSQL to let management software (e.g., Kubernetes operators) switch the primary node to a backup node.

We specify a management interface as a Hoare triple, $\{P\}\ m\ \{Q\}$, meaning that if the precondition P holds before calling the management interface m, then m runs safely and the postcondition Q will hold when m finishes. There is a system-management bug if the operator calls m without ensuring that P holds. We use failover to demonstrate how to specify preconditions and postconditions.

We derive preconditions and postconditions by inspecting the source code of the target system. We start from where the system's management interface is invoked and then inspect how the system validates the arguments and its management state and how it executes the management operation. We focus on the system's successful execution path to derive the precondition that allows the management interface to finish safely and successfully.

Let's apply this procedure to failover. Figure 2a shows the Kubernetes operator selecting a backup node and invoking failover with the current primary and selected backup as inputs. When PostgreSQL receives the request, `is_failover_possible` returns an error if any of the four checks fails as shown in Figure 2b: the primary specified in the request must match the current primary (P_1); in synchronous mode, the backup must be a synchronous standby (P_2); the backup name must identify a member in the current cluster view (P_3); and the backup must pass a promotability check, indicated by `failover_limitation()` returning no limitation (P_4).

Besides the guarding conditions that return errors if unsatisfied, we also consider silent failures when part of the management interface's execution runs asynchronously in the background after it returns. After failover returns, PostgreSQL records the request and wakes its high-availability (HA) loop, which rejects a candidate whose write-ahead-log (WAL) lag exceeds the configured bound in asynchronous replication mode (Figure 2c). So, P_1–P_4 are not sufficient to ensure a safe and successful primary failover. The precondition of failover also includes the bounded-lag check P_5. We

```go
// Select a failover candidate at random.
candidate := replicas[rand.Intn(len(replicas))]
// Send the current primary and candidate to Patroni.
c.Failover(primaryPod, candidate)
```

(a) The management operator selects a candidate and invokes PostgreSQL's failover interface.

```python
# Failover requests use Patroni's failover handler.
def do_POST_failover(self):
    primary, cand = request_inputs()
    error = self.is_failover_possible(cluster, primary, cand)
    if error is None:
        dcs.manual_failover(primary, cand)  # record request
        ha.wakeup()  # notify the HA loop

def is_failover_possible(self, cluster, primary, cand):
    if cluster.primary.name != primary [P1]:
        return ERROR
    if cluster.is_synchronous_mode() and \
       cand not in cluster.sync.members [P2]:
        return ERROR
    members = [m for m in cluster.members if m.name == cand]
    if not members [P3]:
        return ERROR
    for st in fetch_nodes_statuses(members):
        if st.failover_limitation() is not None [P4]:
            return ERROR
    return None  # accept
```

(b) PostgreSQL checks the request and records it for execution.

```python
# Later, the HA loop reads the recorded candidate.
def process_manual_failover_from_leader(self):
    failover = self.cluster.failover
    if is_lagging(status(failover.candidate).wal_position):
        reject_recorded_request()

# Compare the candidate's WAL position with the primary's
    latest WAL position.
def is_lagging(self, candidate_wal_position):
    lag = latest_wal_position - candidate_wal_position
    return lag > maximum_lag_on_failover [P5]
```

(c) The HA loop processes the request and checks replication lag.

Figure 2: Deriving failover preconditions from source code (simplified). Interface inputs are passed from the Kubernetes operator to PostgreSQL.

define the precondition of failover(...) as follows:

$$
\begin{aligned}
P_{\text{failover}} \equiv\ & \text{Primary} = \textit{primary}.\text{Name} \\
& \wedge (\text{sync} \Rightarrow \textit{backup}.\text{Name} \in \text{SyncStandby}) \\
& \wedge \textit{backup}.\text{Name} \in \text{Members} \\
& \wedge \text{Promotable}(\textit{backup}) \\
& \wedge (\text{async} \Rightarrow \textit{backup}.\text{Lag} \leq \text{MaxLagOnFailover})
\end{aligned}
$$

Here, Primary is a variable of PostgreSQL's management state that denotes the primary, sync (resp. async) denotes whether PostgreSQL runs in synchronous (resp. asynchronous) replication mode, Members denotes the list of members, and Promotable(*backup*) means that PostgreSQL reports no failover limitation for the candidate.

Despite the complex precondition, the postcondition for failover() merely states that it returns no error without mentioning any changes to the PostgreSQL membership. The reason is that the membership failover happens asynchronously and there is no guarantee that it is done before failover returns. If the Kubernetes operator invokes failover with all the required preconditions, the only thing the operator can expect after failover returns is that there is no returned error. If the operator wants to ensure the leader failover is done before the next operation, the correct practice is to proactively check the management state of PostgreSQL, e.g., by querying the membership. In fact, there is a common bug pattern that a Kubernetes operator ignores the asynchronous effect and issues the next operation without waiting for the previous operation to (asynchronously) finish [19, 20].

3.2 Writing FOS in P

To write an FOS in P, we construct a *system model* and *invariant monitors*. The system model is written as a state machine where each *action* models how the system handles management interface invocations. The invariant monitors encode the preconditions and postconditions as assertions that are checked before or after each action. Our invariant monitors also check liveness violations, e.g., the operator never receives the response from the system. A key challenge in writing the system model is to choose its abstraction level.

To define the system model state machine, we need to first define the relevant system state. We focus on the management state that is either exposed by the system's management interface or referenced by the interface's preconditions or postconditions. For failover, our system model state machine includes state variables for the current primary, cluster membership, replication mode, the synchronous-standby set, and each member's role, promotion eligibility, and replication lag. We do not model lower-layer components that management interfaces depend on, such as the election protocol, the WAL-streaming pipeline, and other system internals.

We then model each interface's behaviors as actions on the states. Each interface's behavior is represented by handlers that process requests, read or update the management states, and respond to the operator. Figure 3 shows the successful failover path in the system model, which is split into two atomic actions in P. The first action is triggered when there is a pending `eFailoverReq` request sent by the operator, and the system records the request using an abstract `eRunFailover` event and sends back a `eFailoverResp` response to the operator. The `eRunFailover` event later triggers the second

```
on eFailoverReq do (req: tFailoverReq) {
  pending = req;
  send req.client, eFailoverResp, (err = null);
  send app, eRunFailover;
}
on eRunFailover do {
  cand = members[pending.candidate];
  DemoteOldPrimary();
  PromoteCandidate(cand);
}
```

Figure 3: The successful asynchronous failover **path in the Patroni application model.**

```
on eFailoverReq do (req: tFailoverReq) {
  assert req.primary == primary,                                  P1
    "request must name the current primary";
  if (mode == SYNC)
    assert req.candidate in syncStandby,                          P2
      "candidate must be the sync standby";
  assert req.candidate in keys(members),                          P3
    "candidate must be an existing member";
  assert members[req.candidate].promotionEligible,                P4
    "candidate must be promotable";
  if (mode == ASYNC)
    assert members[req.candidate].lag <=
           MAX_LAG_ON_FAILOVER,                                   P5
      "candidate must be within the lag threshold";
}

on eFailoverResp do (resp: tFailoverResp) {
    assert resp.err == null, "failover returned an error";
}
```

Figure 4: Checking the failover **precondition and postcondition in P.** On the request event, the monitor asserts P_1–P_5 (cf. Figure 2); on the response event, it asserts Q_{failover}.

action where the system performs the actual primary failover. The style of splitting the management operation into two actions precisely captures the asynchrony: the primary failover does not necessarily happen when failover returns; it might happen at some later point.

We define the invariant monitors by encoding the preconditions as assertions in P. Figure 4 presents the assertions of P_1–P_5. These assertions will be checked before the first action in Figure 3 is triggered by an `eFailoverReq`. We also encode postconditions as assertions that are checked after this action—the assertions are triggered by an `eFailoverResp` sent by the action. The simple postcondition assertion is mainly used for checking whether the action in P faithfully models the real implementation (see §3.3).

The FOS-based system model and invariant monitors enable automatic and systematic bug detection using P's model checking. If a buggy operator invokes failover without ensuring its precondition, then at least one of the assertions in Figure 4 will fail and P will report a model-checking error.

The precondition assertions check safety, but cannot detect *liveness* violations such as "the operator never makes the managed system enter a desired state" [7, 41]. We are exploring how to encode liveness specifications in P. In principle, detecting liveness violations requires observing an infinite execution to conclude that the desired state *never* happens, which is unrealistic for bounded model checking. P approximates liveness violation detection by checking whether the state machine reaches a user-specified desired state when its model checking terminates. An open question is how to avoid unfair executions during model checking where the P scheduler is unfair and never runs the action that moves the state machine toward the desired state, thus leading to a false liveness violation alarm.

3.3 Checking management correctness

An FOS can be used to systematically check the correctness of the management-plane software—whether the management software follows the formal contract expected by the system. We model the management software as a state machine in P, similar to how we model the managed system, and then compose them into a compound state machine. We then run P's model checking to explore possible executions of the composed model. P reports a safety violation when a precondition assertion fails, and a liveness violation when an execution terminates without reaching the desired state.

The effectiveness of model checking depends on the fidelity of the models, so we check the model-code conformance in two ways. First, we use the postcondition assertions (Figure 4) to check whether each action faithfully models the corresponding code. If the assertion fails, it means the postcondition or the model is wrong. Second, we perform trace validation [10, 16, 21, 32, 42] by collecting implementation-level traces from a real execution of the management software and the managed system, and replaying the traces against the composed P model. The replay checks whether the composed P model can possibly generate the collected traces. The trace validation approximates an implementation-to-model refinement proof that all possible behaviors of the implementation are covered by the model, so the correctness of the model implies the correctness of the implementation.

We have applied this approach to Kubernetes operators. A Kubernetes operator's management logic is implemented as a reconciliation loop, and in each iteration the operator queries the target system's management state and issues corrective management operations (e.g., failover) to the system. We model the operator's reconciliation implementation as a P model. Figure 5 shows a snippet of the PostgreSQL operator's P model, which models the operator code in Figure 2a: the random candidate selection in operator code is modeled by P's nondeterministic `choose`, and the failover call is modeled

```
c = choose(replicas);
send app, eFailoverReq, (primary = primaryPod, candidate = c);
receive eFailoverResp: (r: tFailoverResp);
```

Figure 5: The PostgreSQL operator failover **logic from Figure 2a modeled in P.**

by a send/receive pair: the operator sends eFailoverReq and waits for eFailoverResp.

To check the Kubernetes operator, we also need to model the Kubernetes API. The Kubernetes API represents all identities in the cluster as objects (e.g., Pods) and exposes the objects to operators using REST APIs. For example, a PostgreSQL node is represented as a Pod and the operator queries the status of the PostgreSQL node by reading the state of the Pod. The Kubernetes API is shared by all operators, so we write a reusable Kubernetes API model in P where each state read and write request is an atomic action, following the style of previous work [41]. We compose the FOS of PostgreSQL with the operator model and the Kubernetes API model into a compound state machine, which models how PostgreSQL is managed on a real Kubernetes platform.

Model checking the composed model reveals the bug [35] in Figure 1. P explores an execution where PostgreSQL uses synchronous replication while the operator's choose selects a replica outside its synchronous-standby set, corresponding to the random candidate selection in source code. The invariant monitor's P_2 precondition assertion fails when the model sends the failover request.

4 Preliminary experience and results

Experience. We wrote FOSes for four widely used cloud systems, CockroachDB, Kafka, RabbitMQ, and Solr. We modeled 105 management interfaces in total, such as scaling up and down, node deletions, rolling updates, decommissioning, etc. We wrote 1,946 assertions in our P models. We manually wrote the first FOS (for CockroachDB) by inspecting the source code and summarized our experience and lessons in an agent skill. We then used Claude Code to generate FOSes for other systems. Our experience shows that it is feasible to automate the generation of FOSes using AI agents.

We selected the most popular open-source Kubernetes operator for each of the four systems. Generating the P model for each operator can be mostly automated by Claude Code, because operators' reconciliation patterns are naturally described in P's event-driven model style.

We check each operator using the FOS of the corresponding system managed by the operator. In total, we find 39 new bugs in the four operators. Table 1 summarizes the aforementioned results. We present two new bugs below.

Bug-1: Skipping decommission on downscaling [13]. In the CockroachDB operator, one precondition of its downscaling interface is that the CockroachDB node to be removed must reach the DECOMMISSIONED status. However, the operator does not check this precondition. Such a management operation is unsafe because the removed node is still in the membership as a live member holding replicas. Consequently, the data will not be correctly replicated, leading to potential data loss. P's model checker finds a schedule that violates the precondition, and returns a counterexample.

Pair	Intrf.	Assert.	FOS LOC	KOp LOC	# Bugs
CockroachDB cockroachDB-op.	13	911	11,478	5,031	29
Kafka Strimzi	46	491	11,689	2,743	5
RabbitMQ cluster-op.	8	190	4,648	1,946	3
Solr solr-op.	38	354	10,042	1,761	2
Total	105	1,946	37,857	11,481	**39**

Table 1: Preliminary results of FOSes for four systems, and using them to check Kubernetes operators. Intrf. is the number of interfaces modeled. **Assert.** is the number of assertions. **FOS** and **KOp LOC** show the lines of code in the P models of the FOS and the Kubernetes operator.

Bug-2: Backup hangs after the operator restarts [39]. The Solr operator needs to perform a Solr backup by issuing multiple operations, but it crashes in the middle of its reconciliation, e.g., due to hardware failures. After restarts, the operator cannot recognize the intermediate state left by the previous crash and gets stuck in a state where backup is still in progress but the operator can never finish it—a liveness violation. We define the desired state as "the backup is done," which is never reached by P's model checking.

5 Discussion and open questions

With the recipe of creating the FOS for a given system and its utility of reasoning management correctness, we envision FOS as a part of standard system interfaces. In this section, we discuss how to further automate the generation of FOSes, their use cases for management agents, and their implications for the reliability of system management in general.

Automatically generating FOSes from system code. Our experience (§4) shows that automatically generating an FOS for given system code is feasible and our next step is to develop an agentic tool that leverages coding agents like Claude Code and Codex to realize it. Our plan is to teach AI agents the recipe of writing FOSes (§3) and instruct them to read code, extract invariants, and model the system code from the management-plane perspective. Open questions include: whether agents can carry out these steps reliably, how much human guidance they require, and whether generated FOSes faithfully describe the system implementation (which would require automated trace validation; see §3.3).

Runtime verification of management agents. We have shown how to use FOSes to check the correctness of Kubernetes operators as a common form of management-plane software using model checking. However, the model checking approach does not apply to AI agents which have been increasingly used for production system management [8, 29, 43]. Different from management software, it is hard to write a formal model of AI agents as in §3.3. For management AI agents, the FOS can be used for *runtime verification*—monitoring the runtime behaviors of the agents and checking their proposed management actions against the FOS.

Proving management correctness beyond model checking. We have used model checking to validate the correctness of the management software (§4). However, model checking is known to have scalability issues, especially for system models like the P models we wrote (see [25]). With the FOS, we are exploring how to formally prove the correctness of management-plane software. In fact, P includes PVerifier [34], a deductive backend that proves safety properties from user-provided assumptions and inductive invariants. The open questions are how to reduce the manual proof effort and how to enable liveness proofs. We also plan to port the FOS in P into frameworks like Anvil [41] to build formally verified Kubernetes operators. How to integrate FOS into Anvil's state machine model is a new challenge.

Generating correct-by-construction management operators. A desired use case, suggested by colleagues from the Kubernetes community, is to use FOS to generate Kubernetes operators. Generating an operator requires a management objective in addition to an FOS: the objective states what the operator should achieve, while the FOS constrains how it may invoke management interfaces. Given both, one possible pipeline is to generate the operator as a P model, check or prove it against the FOS, and compile it to executable code [33]. The open questions are whether the objective and FOS provide sufficient information for generating the P model, and how to transform the P model into deployable Kubernetes operators without changing its behaviors.

Improving the reliability and usability of management interfaces of production systems. A long-term solution is to fundamentally enhance the usability and reliability of the management interfaces of production systems. We argue that management interfaces that require complex preconditions are less usable and more error-prone, and have more overhead from the management-plane perspective (as the complexity of router configurations affects network reliability [2, 3, 6]). The FOS can serve as a reference to improve the design of management interfaces.

Connecting FOS to network management. Network management likewise requires management software to understand and respect the semantics of the operations it invokes. For example, taking down a link or upgrading a router should be permitted only when the remaining topology and capacity preserve the deployment's requirements throughout the transition. Configuration management systems such as ADELE use pre- and postconditions to govern configuration changes [18]. Network-management complexity has also been analyzed through structured abstractions such as cross-device dependencies, router roles, reachability constraints, and network-wide policy units [4]. Work on consistent network updates [36] and update synthesis [30] shows how formal specifications can guide such reconfiguration. ScalaCN infers RNIC architecture and performance models from external behavior, suggesting that vendor-written FOSes could avoid such inference, although detailed performance models may expose proprietary information [27]. We advocate making such management semantics an explicit part of the managed device or system's interface through a formal operations specification (FOS): a model of management-relevant state and behavior, together with invariants that constrain management operations. Such contracts would provide a basis for checking management software and agents, and could guide the design of management interfaces.

Acknowledgments

We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their feedback. We also thank Naren Suresh, Shaw Liu, and Tianyu Chen for their participation in the project. This work was funded in part by NSF CNS-2145295 and NSERC RGPIN-2026-06371.

References

[1] Tengfei An and Maria Potop-Butucaru. 2026. A Formal Model of Multi-Controller Orchestration Systems in Cloud Management. In *Proceedings of the 2026 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems (ApPLIED'26)*.

[2] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations. In *Proceedings of the ACM SIGCOMM 2016 Conference (SIGCOMM'16)*.

[3] Theophilus Benson, Aditya Akella, and David Maltz. 2009. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. 335–348.

[4] Theophilus A. Benson. 2012. *New Paradigms for Managing the Complexity and Improving the Performance of Enterprise Networks.* Ph. D. Dissertation. University of Wisconsin–Madison. https://asset.library.wisc.edu/1711.dl/6EEK5ULFYZGUC8M/R/file-ec1d5.pdf

[5] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in

Amazon S3. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*.

[6] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. 2023. Lessons from the Evolution of the Batfish Configuration Analysis Tool. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*.

[7] Zhizhen Cathy Cai, Nikhil Date, Jiawei Tyler Gu, Cody Rivera, Tej Chajed, Oded Padon, Tianyin Xu, and Xudong Sun. 2026. Welder: Compositional Liveness Verification of Cluster Control Planes. In *Proceedings of the 32nd ACM Symposium on Operating Systems Principles (SOSP'26)*.

[8] Yinfang Chen, Jiaqi Pan, Jackson Clark, Yiming Su, Noah Zheutlin, Bhavya Bhavya, Rohan Arora, Yu Deng, Saurabh Jha, and Tianyin Xu. 2025. Stratus: A Multi-agent System for Autonomous Reliability Engineering of Modern Clouds. In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems (NeurIPS'25)*.

[9] Qian Cheng, Ruize Tang, Emilie Ma, Finn Hackett, Peiyang He, Yiming Su, Ivan Beschastnikh, Yu Huang, Xiaoxing Ma, and Tianyin Xu. 2026. SysMoBench: Evaluating AI on Formally Modeling Complex Real-World Systems. In *Proceedings of the 14th International Conference on Learning Representations (ICLR'26)*.

[10] Horatiu Cirstea, Markus A Kuppe, Benjamin Loillier, and Stephan Merz. 2024. Validating Traces of Distributed Programs against TLA+ Specifications. In *Proceedings of the 2024 International Conference on Software Engineering and Formal Methods (SEFM'24)*.

[11] Jackson Clark, Yiming Su, Saad Mohammad Rafid Pial, Yifang Tian, Lily Gniedziejko, Hans-Arno Jacobsen, Yinfang Chen, and Tianyin Xu. 2026. SREGym: A Live Benchmark for AI SRE Agents with High-Fidelity Failure Scenarios. In *Proceedings of the 40th Annual Conference on Neural Information Processing Systems (NeurIPS'26)*.

[12] Cloud Native Computing Foundation. 2021. Operator White Paper. https://www.cncf.io/wp-content/uploads/2021/07/CNCF_Operator_WhitePaper.pdf.

[13] cockroach-operator 2026. cockroach-operator Issue #1151: Scale-down during Failed Rolling Replacement Can Skip Decommission. https://github.com/cockroachdb/cockroach-operator/issues/1151.

[14] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. In *Proceedings of the 35th International Conference on Computer Aided Verification (CAV'23)*.

[15] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe Asynchronous Event-Driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*.

[16] Ding Ding, Zhanghan Wang, Jinyang Li, and Aurojit Panda. 2025. Runtime Protocol Refinement Checking for Distributed Protocol Implementations. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*.

[17] Jason Dobies and Joshua Wood. 2020. *Kubernetes Operators: Automating the Container Orchestration Platform*. O'Reilly Media, Inc.

[18] Jacky Estublier and Rubby Casallas. 1994. The Adele Configuration Manager. In *Configuration Management*, Walter F. Tichy (Ed.). Trends in Software, Vol. 2. John Wiley & Sons, 99–133.

[19] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. 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)*.

[20] Jiawei Tyler Gu, Zhen Tang, Yiming Su, Bogdan Alexandru Stoica, Xudong Sun, William X. Zheng, Yue Zhang, Akond Rahman, Chen Wang, and Tianyin Xu. 2026. Who Watches the Watchers? On the Reliability of Softwarizing Cloud Application Management. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'26)*.

[21] Heidi Howard, Markus A. Kuppe, Edward Ashton, Amaury Chamayou, and Natacha Crooks. 2025. Smart Casual Verification of the Confidential Consortium Framework. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*.

[22] Nouraldin Jaber, Dongyun Jin, Bernhard Kragl, Enrico Magnago, Gustavo Petri, Thorsten Tarrach, and Serdar Tasiran. 2026. High Fidelity Models for Large Scale Stateful Services. In *Proceedings of the 20th USENIX Symposium on Operating Systems Design and Implementation (OSDI'26)*.

[23] Andrew Jeffery and Richard Mortier. 2023. AMC: Towards Trustworthy and Explorable CRDT Applications with the Automerge Model Checker. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'23)*.

[24] Saurabh Jha, Rohan Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, Noah Zheutlin, Saki Takano, Divya Pathak, Felix George, Xinbo Wu, Bekir O. Turkkan, Gerard Vanloo, Michael Nidd, Ting Dai, Oishik Chatterjee, Pranjal Gupta, Suranjana Samanta, Pooja Aggarwal, Rong Lee, Pavankumar Murali, Jae-wook Ahn, Debanjana Kar, Ameet Rahane, Carlos Fonseca, Amit Paradkar, Yu Deng, Pratibha Moogi, Prateeti Mohapatra, Naoki Abe, Chandrasekhar Narayanaswami, Tianyin Xu, Lav R. Varshney, Ruchi Mahindru, Anca Sailer, Laura Shwartz, Daby Sow, Nicholas C. M. Fuller, and Ruchir Puri. 2025. ITBench: Evaluating AI Agents across Diverse Real-World IT Automation Tasks. In *Proceedings of the 42nd International Conference on Machine Learning (ICML'25)*.

[25] Ao Li, Ankush Desai, and Rohan Padhye. 2026. Feedback-guided Adaptive Testing of Distributed Systems Designs. In *Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'26)*.

[26] Bingzhe Liu, Gangmuk Lim, Ryan Beckett, and P. Brighten Godfrey. 2024. Kivi: Verification for Cluster Management. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC'24)*.

[27] Wei Liu, Kun Qian, Zhenhua Li, Feng Qian, Tianyin Xu, Yunhao Liu, Yu Guan, Shuhong Zhu, Hongfei Xu, Lanlan Xi, Chao Qin, and Ennan Zhai. 2025. Mitigating Scalability Walls of RDMA-Based Container Networks. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1049–1065. https://www.usenix.org/conference/nsdi25/presentation/liu-wei

[28] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE'25)*.

[29] Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. 2023. Enhancing Network Management Using Code Generated by Large Language Models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets'23)*.

[30] Jedidiah McClurg, Hossein Hojjat, Pavol Černý, and Nate Foster. 2015. Efficient Synthesis of Network Updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*.

[31] OpenAPI Initiative. [n. d.]. OpenAPI Specification. https://swagger.io/specification/#schema-object.

[32] 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 *Proceedings of the 20th European Conference on Computer Systems (EuroSys'25)*.

[33] P 2022. P2c: Generating C Program. https://p-org.github.io/P/advanced/p2c/.

[34] P 2025. PVerifier: Formal Proofs for P Programs. https://p-org.github.io/P/advanced/PVerifierLanguageExtensions/announcement/.

[35] postgres-operator 2021. postgres-operator Issue #1686: Switchover Fails Randomly in Synchronous Replication. https://github.com/zalando/postgres-operator/issues/1686.

[36] Mark Reitblatt, Nate Foster, Jennifer L. Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference*. 323–334.

[37] Weiming Sheng, Jinlang Wang, Manuel Barros, Aldrin Montana, Jacopo Tagliabue, and Luca Bigon. 2026. GitLake: Git-for-Data for the Agentic Lakehouse. In *Proceedings of the VLDB 2026 Workshop on Systems for Data-Centric Agents with Human-in-the-Loop (DASHSys)*. https://vldb.org/2026/Workshops/VLDB-Workshops-2026/DASHSys/DASHSys26_3.pdf

[38] Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, Suman Nath, Chetan Bansal, and Saravan Rajmohan. 2024. Building AI Agents for Autonomous Clouds: Challenges and Design Principles. In *Proceedings of the 15th ACM Symposium on Cloud Computing (SoCC'24)*.

[39] solr-operator 2026. solr-operator Issue #824: SolrBackup May Never Finish if REQUESTSTATUS Returns notfound for an Accepted Backup. https://github.com/apache/solr-operator/issues/824.

[40] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*.

[41] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*.

[42] Ruize Tang, Minghua Wang, Xudong Sun, Lin Huang, Yu Huang, and Xiaoxing Ma. 2025. Converos: Practical Model Checking for Verifying Rust OS Kernel Concurrency. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC'25)*.

[43] Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, Jerry Yang, and Ying Zhang. 2025. Intent-Driven Network Management with Multi-Agent LLMs: The Confucius Framework. In *Proceedings of the ACM SIGCOMM 2025 Conference (SIGCOMM'25)*.

[44] Qingxin Xu, Yu Gao, and Jun Wei. 2024. An Empirical Study on Kubernetes Operator Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'24)*.

[45] Zalando SE. 2026. Postgres Operator. https://github.com/zalando/postgres-operator.