2025 IEEE 45th International Conference on Distributed Computing Systems (ICDCS) | 979-8-3315-1723-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICDCS63083.2025.00015

Arsc: Automated Reconciliation for State Correctness of Cluster Management

Hongjing Yu†, Chaoqun Li†, Tao Han†, Pengfei Hu†*
† School of Computer Science and Technology, Shandong University, Qingdao, China
{yuhj, chaoqunli, hantaotao}@mail.sdu.edu.cn, phu@sdu.edu.cn

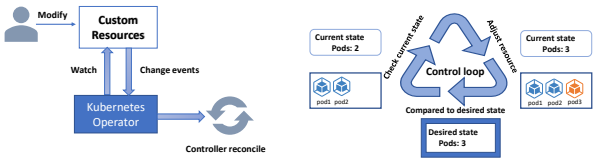
***Abstract*—Modern cluster management platforms, such as Kubernetes, implement declarative interfaces based on state-centric principles and extend cluster capabilities through operators. Once the desired system state is declared, operators automatically reconcile the actual state to align with the desired state. Ensuring state correctness after operations is critical for system safety and stability, as incorrect or unhealthy states can lead to serious consequences. However, there is currently no systematic process in place to guarantee state correctness. The vast state space presents significant challenges in detecting and correcting abnormal states. To tackle this issue, we propose Arsc, an automated technique for test generation, detection, and correction aimed at maintaining state correctness. Arsc establishes the definition of state correctness, offers a state detection method, and presents three strategies for state correction. It was evaluated on five popular open-source operators, generating 11,581 tests and detecting 208 incorrect states, 71 unhealthy states, and 28 crashes, all of which were effectively corrected.**

***Index Terms*—State correctness, Reconciliation, Kubernetes, Operator, System testing**

I. Introduction

With the rapid expansion of cloud computing, the number of nodes in a cluster has grown exponentially, making traditional manual management impractical. Modern cluster management platforms, such as Kubernetes [1], Docker Swarm [2], and Apache Mesos [3], offer high automation, reliability, and continuity. Operators in cluster management extend the cluster's capabilities by automating tasks such as container deployment, networking, and load balancing, significantly improving resource management efficiency and reducing operational costs. Nowadays, the operator pattern is becoming more and more thriving. Taking Kubernetes as an example, an increasing number of commercial and open-source projects have adopted operators to automate various tasks. According to data from OperatorHub, the official Kubernetes-recommended operator community, over 370 operators have been released [4], many of which have been widely deployed in the production environment [5]–[7].

In Kubernetes, operators follow a declarative design pattern with state reconciliation, relying on Custom Resource (CR) [8] and controllers [9]. Operators expose a declarative interface via CRs, which define modifiable properties for managing resources. State declarations specify the property values within CRs. Controllers monitor the resources, and when the state declaration changes, they reconcile the current state to ensure the resources remain in the desired state. Fig.1a briefly illustrates the workflow of an operator, and Fig.1b shows the control loop in the state reconciliation process.



(a) Operator workflow. (b) State reconciliation in controller.

Fig. 1: Operator's working pattern.

The widespread deployment of operators makes ensuring the correctness of their reconciliation results a pressing need. State correctness is crucial for system security and reliability; incorrect or unhealthy states can lead to severe consequences. For example, Fig.2 illustrates an incorrect state in a RabbitMQ cluster. The RabbitMQ operator expects the cluster to have 3 replicas and waits for all replicas to be ready. However, the replica count is overridden to 2 via *statefulSet.spec.replicas*, despite *spec.replicas* being set to 3. As a result, the actual number of replicas does not meet the operator's expectations, preventing progress. More critically, the *StatefulSet* will not be updated until the operator detects 3 replicas, resulting in a persistent incorrect state. Such errors can lead to performance degradation or even serious service outages.

To address state anomalies, users need a reliable method to ensure system state correctness. Most research focuses on improving controller reliability and detecting errors due to operator design flaws [10]–[12]. In contrast, our work aims to develop a technique for detecting and correcting state anomalies that occur during operator operation. These anomalies can arise from various sources, not just potential flaws in the operator itself. We present Arsc, an automated tool for ensuring state correctness in cluster management. Arsc generates state declarations to guide operators through the workflow and monitors the system state during operation. If anomalies are detected, Arsc takes corrective actions. State correctness is defined by three criteria: (1) conformity with the desired state, (2) a healthy state, and (3) if the current state is abnormal, returning at least to the previous correct state.

Arsc consists of three main functional modules: the generation module, the detection module, and the correction module.

* Pengfei Hu is the corresponding author.

2575-8411/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDCS63083.2025.00015

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

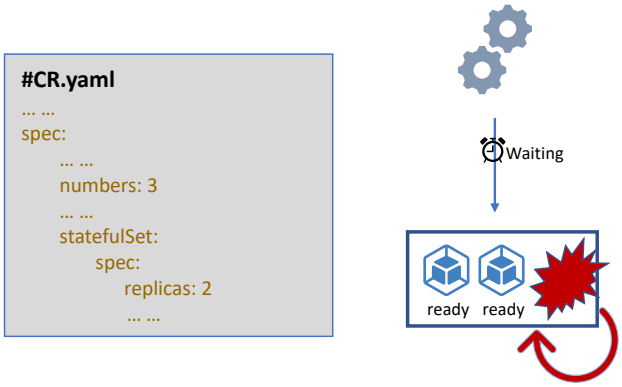


Fig. 2: An incorrect state in RabbitMQ cluster.

In the generation module, Arsc creates state declarations by parsing the Custom Resource Definition (CRD), which defines property constraints in the *OpenAPISchema* format, to explore state transitions in different scenarios. In the detection module, Arsc detects whether the system state is correct. The complex state space presents a challenge for us. We focus on system resources, particularly CRs, and establish the definition of state correctness along with a method for state detection. In the correction module, Arsc corrects detected anomalies in three ways. State recovery fixes issues caused by failed adjustments or temporary faults by modifying resources or configurations. State rollback handles anomalies from invalid declarations or irrecoverable failures by reverting to a previous state. Restart restores the system to its initial state after a crash. The operator continuously reconciles the system state based on a sequence of generated state declarations. Each state declaration drives the operator to transition the system to a new state, from which the subsequent operation begins. This process leverages the level-triggering principle [13].

We evaluated Arsc on five popular Kubernetes operators. Arsc generates 11581 state declarations and detects 208 incorrect states, 71 unhealthy states, and 28 crashes. Arsc was able to effectively detect system anomalies and perform reliable correction.

Contributions. The main contributions of this paper are summarized as follows:

- We present an automated technique that ensures state correctness for cloud system operators.
- We implement Arsc, an automated generation, detection, and correction tool based on state-centric principle.
- Arsc has already been evaluated on five Kubernetes operators. The results show it is reliable.

II. Related Work

In this section, we provide an overview of related work in the areas of controller correctness, system verification, cluster management verification, and state space exploration. These fields are crucial for understanding the current approaches and techniques used to ensure the reliability, correctness, and efficiency of distributed systems, particularly in the context of Kubernetes.

A. Controller Correctness

As part of a literature review on controller correctness and verification techniques, several notable works have been identified that contribute to the field. Sun et al. [10] employed fault injection techniques to verify the reliability of controller implementations, aiming to identify potential faults. Gu et al. [12] adopted an end-to-end testing approach to evaluate the correctness of controllers. Liu et al. [11] utilized model checking to verify controllers and their configurations. In a broader context, Sun et al. [14] proposed a general controller correctness specification framework to formalize the process of verifying controller implementations, providing guidelines for ensuring that controllers align with their intended functionality and meet the requirements of distributed systems. These works primarily focus on identifying potential flaws within the controller itself, while our approach examines the system to detect state anomalies. These anomalies may arise from various factors, including but not limited to potential flaws in controllers.

B. System Verification

Verification techniques have been successfully applied to various distributed systems [15]–[23] and network protocols [24]–[28]. For example, Yabandeh et al. [29] used model checking methods to prevent inconsistencies in distributed systems by analyzing possible state transitions. Hawblitzel et al. [30] focused on system liveness. While Kubernetes shares the characteristics of distributed systems, existing works cannot be directly applied. Kubernetes tends to focus on verifying resource management and scheduling, whereas other distributed systems may prioritize properties such as data consistency or fault tolerance. Moreover, the resource states in Kubernetes differ significantly, as they involve the dynamic management of large-scale systems with complex dependencies.

C. Cluster Management Verification

Formal verification methods have also been applied to cluster management systems to ensure the correctness of their operational behavior. Turin et al. [31] presented a formal model of Kubernetes based on simplified assumptions of controllers but did not test it for verification. Liu et al. [32] proposed a proof-of-concept method, focusing on modeling several failure scenarios. Ding et al. [33] applied verification techniques to serverless architectures, with a particular focus on idempotence properties from the application perspective. Our work focuses on state, particularly resource state, and aims to detect system state in real Kubernetes working environments.

D. State Space Exploration

Exploring the state space of complex systems is a challenging task due to its vastness and complexity. Godefroid [34] proposed a method that divides the state space into

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

local and global states, which reduces the complexity of state space search and makes it more feasible to explore large-scale systems. Yang et al. [35] applied state space reduction techniques to storage systems. Yabandeh et al. [29] used distributed snapshots to explore the state space to a certain depth from an intermediate state during runtime, checking whether the system state violates a given property. The state space in Kubernetes is complex, and Godefroid's work has provided us with valuable insights. Based on the operational characteristics of operators, after state reconciliation, we focus on CRs and the dependent resources related to them.

III. Motivation

The operator pattern flourishes. The scalability of Kubernetes has fostered a thriving ecosystem of thousands of domain-specific operators [36]–[39]. Thousands of operators developed by both commercial vendors and the open-source community. For instance, OperatorHub [4] offers over 370 operators, and ArtifactHub [40] offers over 1000 operators. While these technological advancements provide significant convenience and flexibility, they also introduce new challenges. As the Kubernetes ecosystem continues to grow, ensuring the state correctness has become a key issue that needs to be addressed.

Kubernetes clusters are state-centric. The functionality and stability of the system depend on the continuous management and synchronization of its current state. We analyzed 64 Common Vulnerabilities and Exposures (CVEs) listed in the official Kubernetes Security Response Committee and found that most directly affect the system state, with two specifically related to CRs [41]. Any deviation between the actual and desired states can lead to service disruptions or even cause the entire system to crash. This highlights the importance of effective state management and correction mechanisms as the cornerstone for maintaining a reliable and resilient Kubernetes environment.

The need for comprehensive method. Existing research primarily focuses on identifying vulnerabilities or implementation issues within the operator itself. In contrast, there is a lack of methods focused on detecting and ensuring the correctness of Kubernetes cluster state. By monitoring the dynamic changes of the cluster state, we can safeguard the system's stability and reliability at a broader level, rather than solely concentrating on the security of individual operators.

IV. Technique

Arsc is committed to maintaining the correctness of the system state by monitoring system resources. It detects anomalies when state correctness requirements are violated and takes action to reconcile the system. Such anomalies include: (1) not meeting the desired state, (2) being in an unhealthy state, and (3) a system crash may occur under special circumstances.

To use Arsc, users need to provide the configuration information of the operator, guiding Arsc to correctly deploy the operator. Arsc accepts CRD as input and ultimately provides detecting and correction results. The workflow of Arsc is described in section IV-A.

A. Workflow

As shown in Fig. 3, Arsc consists of three main modules: the generation module, the detection module, and the correction module. Its workflow can be summarized in seven steps: ① deploying the operator, ② parsing the CRD, ③ generating state declarations, ④ reconciling the state, ⑤ detecting whether the system state is correct, ⑥ correcting the state, and ⑦ collecting the results. The following describes the five main components of Arsc:

Parser: The parser takes the CRD file as input, parses it into the target format, and performs both syntactic and semantic analysis. It extracts key properties and analyzes their usage scenarios. As one of the core components for generation module, it provides a foundation for targeted state declaration generation through structural parsing and semantic analysis of the CRD.

Generator: The generator automatically creates CR files that declare the desired state based on the parser's results. We ensure that the generated state declarations are syntactically correct, but not necessarily semantically correct. In other words, in addition to valid operation scenarios, the generator also generates invalid declarations to test whether Arsc can detect and correct state anomalies in time. The process of generating state declarations is described in detail in section IV-B.

Monitor: The monitor is responsible for real-time monitoring of the system state. It interacts with the Kubernetes API to regularly collect system operation data, such as resource states, system events, etc. Once the system state is deemed stable, it captures a system state snapshot for subsequent state detection and correction. The key point is explained in section IV-C.

Checker: The checker compares and checks the differences between the current system state and the desired state. Its inputs include the real-time system state obtained from the monitor and the desired state provided by the generated state declaration. In section IV-D, we establish the definition of state correctness and describe how to detect resource state.

Mediator: The mediator is the key correction component of Arsc, responsible for attempting to restore the system to a correct state by automating resource adjustments when system anomalies are detected. The mediator provides three strategies, with the goal of achieving "self-healing", which are explained in section IV-E.

In short, the parser and generator are responsible for generating CR files which provide state declarations, while the monitor and checker handle system state detection. The checker and mediator are tasked with correcting state anomalies.

B. CR Files Generation

1) CRD Parsing: The core task of a Kubernetes operator is to automatically manage the cluster based on the state declarations of CRs. CRD defines the structure, fields, and validation

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

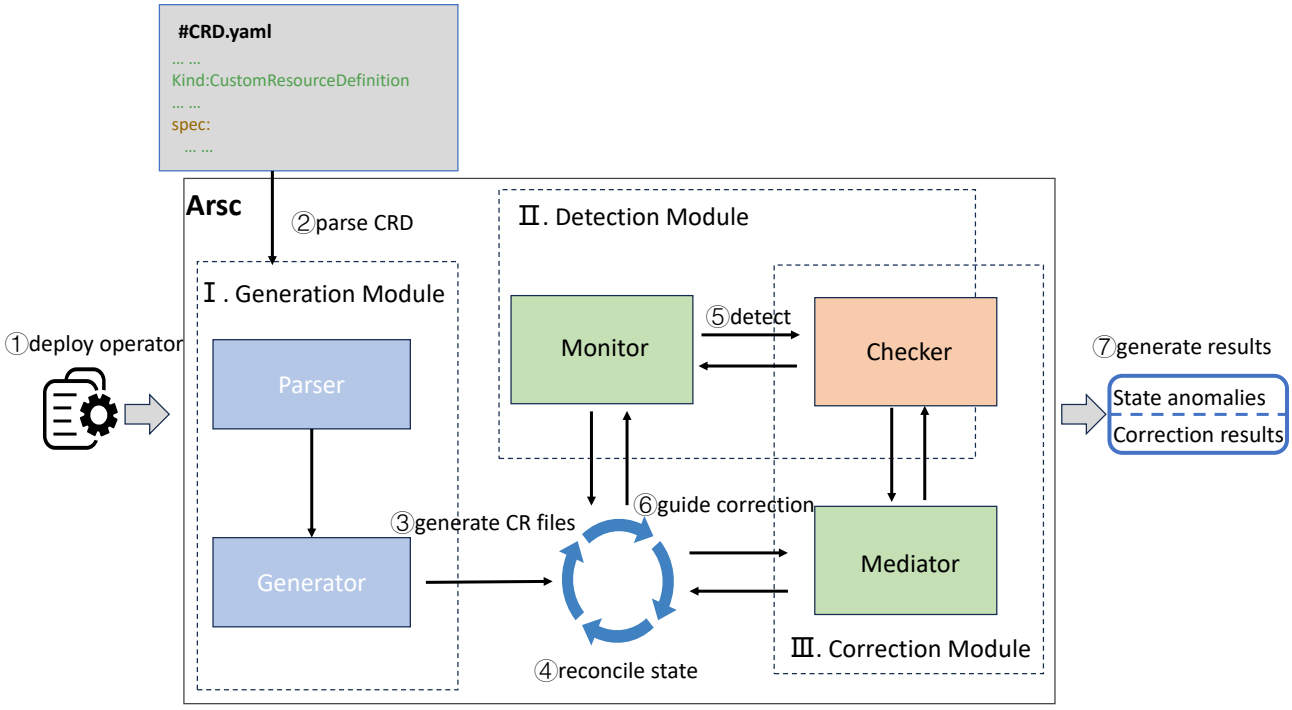


Fig. 3: The workflow of our proposed Arsc method.

rules of CRs. The parser analyzes the CRD files, extracting structural and property information, while the generator creates the CR files. The operator monitors changes in the CRs and performs the corresponding actions based on those changes.

Syntax Analysis: The CRD's *spec.openAPI3Schema* specification defines constraints for all supported properties. The parser extracts these constraints to ensure that all property values are syntactically valid. For simple properties and basic data types (e.g., *type: string*), the type constraints, value range limitations, and format requirements are extracted, and the dependencies between fields are recorded. For nested properties and complex data types (e.g., *type: object*), each possible combination of conditions is expanded, generating multiple independent branches. For special keywords (e.g., *anyOf*, *oneOf*, *allOf*), if complex cross-field constraints exist, corresponding constraint expressions are generated and marked as logical points. In practice, the structure of CRD files in real-world is often much more complex, resulting in a large number of branches during the search process. To minimize complexity, we define pruning strategies that terminate the search early for branches determined to be redundant (e.g., *description*). We also employ heuristic searches to prioritize fields that significantly impact operator behavior (e.g., *replicas*, *scale*, *timeout*, etc.).

Semantic Analysis: Syntax analysis provides only basic information, such as the data type and the constraints of properties. Obtaining the functionality and usage scenarios requires further semantic analysis. The most straightforward method for semantic inference is to deduce from the field name. For example, common naming patterns, such as *minXXX* or *maxXXX*, can provide hints during semantic inference. Additionally, we create a mapping table based on Kubernetes terminology, which helps us infer the potential semantics by matching them with Kubernetes terms.

The actual semantics of fields may be ambiguous due to differences in naming conventions, domain knowledge, and context. Moreover, CRD is static and does not directly reflect how an operator dynamically adjusts resources at runtime. By tracking the data flow of a property, we can determine how the property value is used. If a property value is passed to the Kubernetes API or assigned to Kubernetes resource objects, we use the mapping table to map the property to the corresponding Kubernetes object. System state changes often involve aspects such as replica counts, resource configuration, and lifecycle management. We focus on properties relevant to these scenarios.

2) Generating Strategy: Based on the results of syntax and semantic analysis, all properties are assigned values using different strategies depending on their *type*. The generator focuses on exploring various scenarios based on semantics, with each property value representing a specific scenario. For properties where semantic analysis fails, the generator generates property values according to syntactic rules. These values may be semantically valid or invalid.

C. State Transition

1) State Reconciliation: A series of CR files are generated and sequentially applied to the cluster. The operator

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

continuously monitors resource change events. When a state declaration is applied to the cluster, the operator receives an event notification. It then captures the current state of the cluster resources, analyzes the desired state described in the state declaration, and generates a reconciliation plan based on the differences. Finally, the operator invokes the corresponding resources through the Kubernetes API and performs the necessary operations to bring the current state into alignment with the desired state.

2) Capturing System State: To detect system state, the monitor needs to capture a system-wide snapshot in order to fully understand the state of the cluster.

Checkpoint: Ideally, the snapshot captured after the operator reconciliation should represent the up-to-date and stable state, rather than an intermediate or incomplete state during the process. Therefore, the timing of checkpoint selection is crucial. Capturing the system state too early may lead to false positives, as the system has not fully completed reconciliation. Capturing it too late may waste resources or miss critical points of fluctuation in the system state. To optimize this process, we drew on the work of Sun et al., Anvil [14], which introduced the concept of Eventually Stable Reconciliation (ESR). ESR focuses on two core concepts: progress and stability, asserting that the system should reconcile the cluster's state to meet the desired state and maintain that state after reconciliation is complete. We adapted this concept by using a dynamic adjustment mechanism to select checkpoints, employing delay injection to capture the stable state after reconciliation.

System Snapshot: When capturing a snapshot, the focus is on resources (e.g., *Deployments*, *Pods*, *Services*) and external dependencies (e.g., *ConfigMaps* and *Secrets*). Information is recorded at multiple levels, including resource metadata, runtime status, and interactions with external dependencies.

D. State Detection

1) State Correctness Definition: In Kubernetes, resources refer to the various objects used for management and scheduling within the cluster. Each resource has specific functions and properties, and Kubernetes uses these resources to achieve automated management and reconciliation of the cluster. The state of a resource reflects its current configuration and operational status. The system state provides an overall description of these resource configurations. Tab. I lists the common types of resources in Kubernetes, including native and custom resources. These resource types form the foundation of Kubernetes management and scheduling.

Kubernetes resources often depend on each other, where the proper functioning of one resource relies on the state of another. For example, a pod may depend on a configMap for configuration, or a service may rely on a deployment to manage pods. Understanding these dependencies is crucial for grasping how resource states impact each other in the cluster.

We can represent the system's resources as a set $R = \{R_1, R_2, \ldots, R_n\}$, where $R_i \in R$ $(1 \leq i \leq n)$. The dependency relationships between these resources are defined as E, such that if R_i depends on R_j, then $(R_i \to R_j) \in E$.

The system state is denoted as $S(R)$, where $s(R_i)$ represents the actual state of resource R_i, and the desired state is denoted as $D(R)$, where $d(R_i)$ represents the desired state of resource R_i.

Definition 1. *Given the system state $S(R)$ and the desired state $D(R)$, we say that S matches D, denoted as $S \sim D$, if the following conditions are satisfied:*

- *For each resource $R_i \in R$, the actual state $s(R_i)$ must match the desired state $d(R_i)$.*
- *For each dependency $(R_i \to R_j) \in E$, the actual state $s(R_j)$ of the dependent resource R_j must satisfy the requirements set by R_i.*

Formally, this consistency is defined as:

$$S \sim D \iff \forall R_i \in R,\ s(R_i) = d(R_i) \wedge$$
$$\forall (R_i \to R_j) \in E,\ s(R_j) \text{ satisfies the requirements of } R_i.$$

After establishing the consistency between the system state and the desired state ($S \sim D$), it is essential to address the health of the system's resources. While $S \sim D$ ensures that the actual state of the resources aligns with the desired state, it does not necessarily guarantee that the resources are functioning optimally. The health of the resources, defined by the "SAR" conditions, plays a critical role in ensuring the stability, availability, and responsiveness of the system.

Definition 2. *For any $R_i \in R$, the system state is considered healthy, denoted as $S \sim H$, if $s(R_i)$ satisfies the "SAR" conditions, where:*

- *Stable: The resource remains stable without frequent restarts or failures.*
- *Available: The resource is ready to accept requests and provide services.*
- *Responsive: The resource responds to traffic or requests in a timely manner, ensuring good service quality.*

After defining the consistency between the system state and the desired state ($S \sim D$), as well as the health condition of the resources ($S \sim H$), we can conclude that the system state is correct if both conditions are satisfied.

Definition 3. *If $S \sim D$ and $S \sim H$, then the system state is considered correct, denoted as $S \sim C$.*

2) Local Detection: Local detection focuses on detecting the state correctness by checking only the custom resources and their dependent resources, rather than examining the entire set of system resources. This approach reduces the scope of the detection process, making it more efficient and scalable.

The core idea behind local detection is that the system's behavior and overall state are primarily determined by the custom resources and the resources that directly interact with them, particularly in the context of the operator pattern. By narrowing the detection scope to these resources, we can capture inconsistencies between the actual and desired system

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Some resources in cluster.

Kind	Resource	Description
Computing resource	Pod	Pod is the smallest unit of compute in Kubernetes and usually consists of one or more tightly coupled containers.
	Deployment	Deployment is a declarative resource used to manage the lifecycle of Pods.
	StatefulSet	StatefulSet is used for managing stateful applications like databases.
	DaemonSet	DaemonSet ensures that a copy of a Pod is running on each node in the cluster.
Storage resource	PersistentVolume	PersistentVolume abstracts physical storage devices and manages the lifecycle of storage volumes.
	PersistentVolumeClaim	PVC is a request for storage resources by a Pod.
	ConfigMap	ConfigMap is a resource used to store non-sensitive configuration data that can be shared across multiple Pods.
	Secret	Secret is a resource for storing sensitive data like passwords, API keys, etc.
Network resource	Service	Service is a Kubernetes network resource used to expose a set of Pods and provide load balancing.
	Ingress	Ingress manages external HTTP and HTTPS traffic to services in the cluster.
	NetworkPolicy	NetworkPolicy defines the rules for network communication between Pods, controlling which Pods can communicate and which cannot.
Scheduling and management	Namespace	Namespace is a mechanism for dividing resources into logical groups within a Kubernetes cluster.
	Node	Node is a worker machine in Kubernetes, responsible for running containerized workloads.
	ResourceQuota	ResourceQuota limits the total amount of resources that can be used within a namespace, preventing resource overuse.
Custom resource	-	Custom resource extends Kubernetes by defining my own resource types.

states, while reducing the overhead of checking irrelevant resources.

The process can be broken down as follows:

- Identify the custom resources: The primary focus is on the custom resources, which represent user-defined objects and configurations in the Kubernetes system.
- Determine dependencies: Identify the native resources that the custom resource depends on, such as *Pod*, *Service*, *ConfigMap*, or other Kubernetes objects.
- Verify state consistency: For the custom resources and its dependencies, check if the actual state matches the desired state, as defined by the state specifications. Also, verify the health status of the resources.
- Detect anomalies locally: Any inconsistencies or unhealthiness found in the custom resources or its dependent resources are flagged for further action.

3) Global Detection: Global detection focuses on detecting the health of system resources that are independent of custom resources. It can be roughly assumed that the state of resources not mentioned in the state declaration remains stable and only needs to be checked for compliance with Definition 2.

This assumption is based on the premise that the system starts from a healthy state. The cluster configuration is typically rigorously validated by the vendor during the deployment phase, which minimizes the risk of errors or inconsistencies in the initial state. Since the system is deployed with a healthy and validated configuration, the primary focus of detection shifts to ensuring the ongoing health of native resources within the cluster. Therefore, global detection emphasizes detecting whether resources remain stable, available, and responsive over time.

4) Detection Method: In practice, both detection methods are applied simultaneously, complementing each other to provide a comprehensive solution for system state detecting. Local detection focuses on detecting custom resources and their direct dependencies, offering a more targeted approach to ensuring state consistency within the operator's scope. Global detection, on the other hand, shifts its focus to maintaining the ongoing health and stability of native resources, enabling monitoring of the system's overall state.

E. State Correction

1) State Recovery: In the case of simple system anomalies, the first step is to attempt reconciliation in the current state. Upon detecting an anomaly, the mediator triggers event filtering, which serves as the system's immediate response mechanism. This step helps prevent misoperations and limits the propagation of errors. Event filtering is implemented by inserting hooks into the event processing flow, which are supplemented by timer-based delay logic.

Event filtering provides the system with sufficient time to recover and return to the correct state. After event filtering, the mediator initiates corrective actions to restore the system to the desired state. These corrective actions heavily rely on the controller's self-healing capabilities, particularly its built-in control loop. Such actions include resource adjustment, rescheduling, and fixing configuration errors, and more. The process is illustrated in Fig. 4.

2) State Rollback: If the state recovery mechanism fails to reconcile the system to a correct state, the rollback mechanism is triggered to restore the system to the last known correct state. This rollback mechanism ensures a "safe starting point" for subsequent detection and intervention. The process follows these steps: (1) When the system state is found to be abnormal and cannot be corrected through simpler recovery operations, the rollback mechanism identifies the most recent valid state snapshot. (2) It then restores critical components to this known state, clearing any operations or data that may have caused the current inconsistencies. (3) Once the rollback is complete, the

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

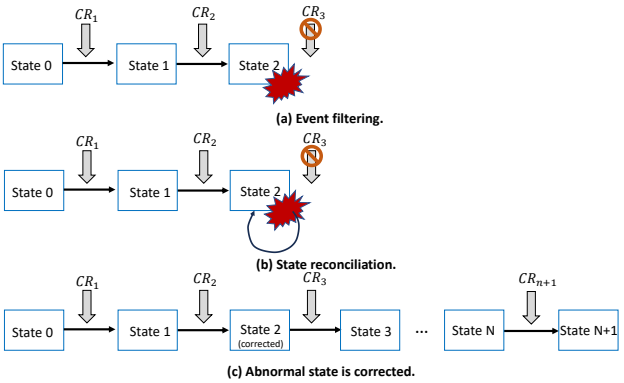


Fig. 4: State recovery process.

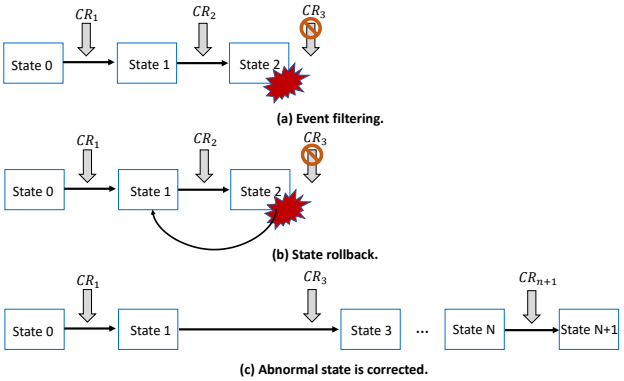


Fig. 5: State rollback process.

system resumes operation from the restored state and proceeds with further operations. The rollback process is shown in Fig. 5.

3) Restart: In cases where the system encounters an unrecoverable crash due to a critical error, a full restart from the initial state is required. This ensures that the system starts fresh, eliminating the possibility of residual inconsistencies. The restart process is depicted in Fig. 6.

4) Strategy Priority: State recovery is suitable for state anomalies that can be resolved through resource adjustments and similar measures. State rollback is used for irrecoverable state anomalies. Restart is employed in the event of a system

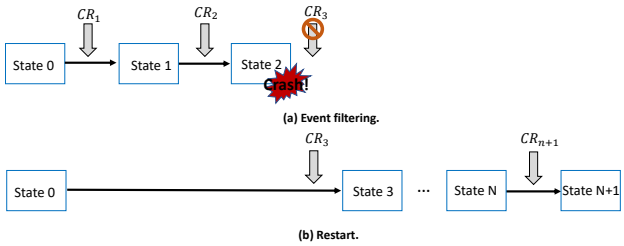


Fig. 6: Restart process.

crash. The anomalies detected by the detection module can be categorized into three types: incorrect state, unhealthy state, and system crash. After detecting a state anomaly, selecting the appropriate correction strategy is crucial to ensuring system stability. We prioritize three correction strategies in the following order: state recovery, state rollback, and restart. For both incorrect and unhealthy states, the state recovery strategy is applied first. If state recovery is insufficient, the state rollback strategy is used. In the case of system crashes, the restart strategy is typically employed.

5) State Recheck: To ensure the correctness and effectiveness of the correction process, the system undergoes comprehensive revalidation after any corrective action. This revalidation includes checking the state of individual components, verifying the consistency of inter-component dependencies, and confirming overall system health. The recheck process not only ensures that the system has returned to the desired healthy state but also verifies that business logic is functioning as expected and that previous abnormalities have been successfully resolved.

F. Implementation

Arsc is designed to ensure state correctness in cluster management. It operates on a virtualized Kubernetes cluster. Its primary focus is the analysis of system resources, offering a degree of generality. Arsc executes tasks sequentially: when the Kubernetes API receives a new state declaration, the operator compares the current state with the desired state and makes the necessary adjustments. Once Arsc confirms that the system state is correct, the Kubernetes API can receive the next state declaration, meaning the current state serves as the starting point for the subsequent state.

V. Evaluation

Our experiments were conducted on a Cambricon server equipped with two Intel(R) Xeon(R) Platinum 8358 CPUs (32 cores) and 503GB of memory, running Ubuntu 22.04.3 LTS with kernel version 5.15.0-119-generic. We used Kind [42], a tool designed to run Kubernetes clusters within Docker containers, to quickly deploy the Kubernetes cluster. Considering system load, we configured only one worker node for the experiment.

To evaluate Arsc, we tested it with five popular open-source Kubernetes operators, all of which were developed either by the official teams of the managed systems or by companies offering services built around these systems (Tab. II). We focus on three key questions: (1) Can Arsc generate efficient state declarations? (2) Can Arsc detect system anomalies? (3) After detecting anomalies, can Arsc successfully correct the system? The main evaluation results are summarized below:

- Arsc generates 11,581 state declarations, and after analysis, 11387 are valid and 194 are invalid.
- Arsc detects 208 incorrect states, 71 unhealthy states, and 28 crashes.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

- Arsc applies different correction strategies based on the types of anomalies and ensures effective correction for all detected state anomalies.

TABLE II: The operators we used.

Operator	Provider	Categories
RocketMQ	Apache Software Foundation	Streaming & Messaging
Cockroach	Cockroach Labs	Database
CloudnativePG	CloudNativePG Contributors	Database
Zookeeper	StreamNative	Storage
RabbitMQ	VMware Tanzu	Streaming & Messaging

A. Generation Results

1) Validity of Declaration: Valid state declarations clearly and accurately describe the desired state, with parameters and configurations strictly adhering to the defined specifications, enabling the system to correctly parse and execute them. Specifically, the replica counts, resource requests, and other parameters in the generated CR files conform to the CRD requirements, allowing the system to allocate resources and achieve the desired state based on these valid declarations. In contrast, invalid state declarations fail to meet expectations or rules, possibly due to configuration errors, out-of-range parameter values, or violations of constraints, rendering them unrecognizable or unprocessable by the system. The generator generated 11,581 state declarations, of which 11387 were valid and 194 were invalid. We present the generation results in Tab. III.

Invalid declarations can lead to incorrect or unhealthy states and may even cause system malfunctions or crashes. In the correction process, we chose to skip these invalid declarations to maintain system correctness.

TABLE III: Generation results.

	Type	Num	Percentage
Valid	-	11387	98.32%
Invalid	Invalid field	43	0.37%
	Invalid value	65	0.56%
	Invalid scale	37	0.32%
	Invalid label	35	0.30%
	Others	14	0.12%
	Total	11581	-

2) Property Coverage: Arsc achieves 100% property coverage for each operator, with the generator ensuring at least one operation for every property. The automated generation of state declarations not only covers a wide range of properties but also generates more property values and implements a greater number of state transitions.

B. Detection Results

Arsc detects 208 incorrect states, 71 unhealthy states, and 28 crashes. Detailed results are provided in Tab. IV.

1) Incorrect State: Arsc identified 208 instances of incorrect states that violated state declaration. An incorrect state refers to a discrepancy between the system's actual state and its desired state. This does not imply that the system is completely crashed or unhealthy, but rather indicates issues such as configuration errors, resource allocation problems, or state declaration exceptions, which prevent resources or system behavior from meeting the desired objectives.

TABLE IV: Detection results.

Operator	Test	Incorrect	Unhealthy	Crash
RocketMQ	1584	27	0	0
Cockroach	737	29	9	12
CloudnativePG	2093	32	40	6
Zookeeper	2880	79	16	5
RabbitMQ	4287	41	6	5
Total	11581	208	71	28

Case 1: An issue was observed in the RabbitMQ cluster, which has been mentioned in fig. 2. The number of replicas is overridden to 2 via *statefulSet.spec.replicas*, even though *spec.replicas* is set to 3. After the system scales to two replicas, the operator stops reconciling, causing the number of replicas to no longer match the desired state.

Case 2: This is an issue found in the RocketMQ cluster where incorrect states were caused by the inclusion of invalid fields. The *spec.resources* field is used to define resource requests and limits for containers, typically including standard resource types such as CPU, memory, and storage. These fields have strict format requirements to ensure accurate and consistent resource allocation. However, adding invalid fields in the resources configuration prevents Kubernetes' internal resource management system from recognizing them, leading to inaccurate resource allocation.

2) Unhealthy State: Arsc identified 71 instances of unhealthy states. An unhealthy state refers to a condition where the system or resource is running normally, but its health status does not meet expectations, potentially indicating underlying faults or instability.

Case 1: This issue was discovered in the RocketMQ cluster, caused by the use of invalid values in the configuration fields. The *spec.env* field is used to define environment variables for containers, and the values can be set using either the *value* or *valueFrom* field. However, *value* and *valueFrom* are mutually exclusive fields. If both are set for the same environment variable, it results in a configuration conflict. The Kubernetes API server returns a validation error, indicating the field conflict and refusing to apply the configuration.

Case 2: A pod startup failure was observed in the RabbitMQ cluster. The cluster added two pods and their corresponding persistent volumes through the state declaration. However, due to false binding of dependencies, both newly added pods remained in the Pending state.

3) Crash: Arsc identified 28 instances of crashes. A crash refers to a complete failure of the system or service, rendering it unable to operate normally. System crashes typically occur when a process or node fails, causing part or all of the system to be unable to provide services.

Case 1: This crash scenario was observed in a Cloudnative cluster. When the *ephemeralVolumesSizeLimit* is not specified, it indicates that there is no explicit limit on the ephemeral storage size for PostgreSQL cluster instances. However, when

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

the shared memory size is explicitly configured to 4Gi using *ephemeralVolumesSizeLimit.shm*, it sets a 4Gi limit on shared memory for each PostgreSQL cluster instance. Under this constraint, attempting to allocate memory beyond the shared memory limit results in a system crash.

Case 2: This crash scenario occurred in a Zookeeper cluster. The *pod.securityContext* field configures the security context of a pod, including several key settings: *fsGroup* defines the filesystem group inside the container, *runAsGroup* sets the process group, *runAsUser* defines the user ID for the process, and *supplementalGroups* specifies additional group IDs. The crash was caused by an incompatibility between the *pod.securityContext* configuration and the filesystem permissions, storage volume access, and cluster security policies.

4) False Positive: A total of 194 state anomalies were caused by invalid state declarations, while 113 resulted from other factors. Additionally, we identified one false positive in RabbitMQ, caused by the unclear impact of the *skipPostDeploySteps* property on state. Therefore, the overall false positive rate of Arsc is 0.32%.

C. Correction Results

Once a state anomaly is detected, the correction module selects the state recovery, state rollback, or restart solution in order of priority. The correction results for the three types of state anomalies are shown in Tab. V.

TABLE V: The correction rate of abnormal state.

Strategy	Incorrect	Unhealthy	Crash
State recovery	11.06%	87.32%	-
State rollback	88.94%	12.68%	-
Restart	-	-	100%

1) Correction for Incorrect State: For incorrect states, the state recovery strategy is prioritized, achieving a correction rate of 11.06%. For states that cannot be corrected by the state recovery strategy, the state rollback strategy is applied.

The incorrect states identified in our experiments fall into two main types: The first type is adjustment failure, where the system fails to adjust the resource state to match the desired state, as illustrated in Case 1 in section V-B1. For this type of incorrect state, we apply the state recovery strategy, which restores the system to the desired state by re-adjusting the resources. The second type is invalid state declarations, where users may incorrectly declare resource configurations in custom resources, leading the system to believe the resources are correctly configured when, in fact, they do not meet expectations. Our experiments show that this type of error constitutes the majority of incorrect states, approximately 88.94%. In this case, we correct the anomaly by rolling back to the previous correct state and skipping the invalid declaration. This approach not only proved effective in experiments but also offers practical insights for real-world applications. In practice, invalid state declarations are almost inevitable, and therefore, employing effective strategies to skip these declarations and avoid potential errors is essential.

2) Correction for Unhealthy State: The causes of unhealthy states are primarily due to insufficient system resources, configuration errors, and external dependency failures. To address these issues, the state recovery strategy adjusts resource limits, optimizes resource allocation, and waits for external services when necessary. For irrecoverable unhealthy states, such as Case 1 in the section V-B2, the state rollback strategy is applied. The correction strategy for unhealthy states is similar to that for incorrect states.The state recovery strategy can correct 87.32% of unhealthy states, while some states, due to invalid state declarations or other special reasons, may require the state rollback strategy.

3) Correction for System Crash: Restarting resolved all identified crash issues. When a system crashes, it is typically due to a severe error or exceptional condition that prevents recovery or continued operation. Compared to attempting more complex recovery methods, restarting is the simplest and most effective solution. It immediately restarts the process, ensuring the service begins from its initial state, reloads all configurations, and resumes normal operation. This method is universally efficient, but we do not advocate for its use as a preferred solution.

4) Reflection: The current correction strategies ensure that the system maintains the correct state, but in some cases, they may lead to the loss of certain progress, particularly in the case of meaningless state declarations. This is unavoidable, and we believe that skipping such operations is acceptable.

D. Execution Time

We deployed five open-source operators and conducted evaluation experiments on their respective clusters. As shown in Tab. VI, the total execution time ranges from 22.53 to 133.66 hours across the operators. The majority of Arsc's runtime is spent on detection and correction. The experimental duration varies significantly due to the differing operational complexity of each operator.

TABLE VI: Arsc execution time for per operator (hours).

Operator	Generation	Detection & Correction	Total
RocketMQ	0.01	22.52	22.53
Cockroach	0.01	97.28	97.29
CloudnativePG	0.02	95.34	95.36
Zookeeper	0.02	136.64	136.66
RabbitMQ	0.03	75.04	75.07

VI. Limitations and Future Work

Limitation in scenario coverage. Modern cloud cluster systems consist of a wide range of components and resources, leading to a vast and highly dynamic state space. Due to the complex interactions among these components, system anomalies can manifest in numerous patterns. As a result, abnormal events are not only difficult to predict but also challenging to reproduce in a controlled environment. When generating test cases, the complexity and variability of the state space make it difficult to ensure comprehensive coverage of all potential failure scenarios.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

Limitation in integrality. Intermediate anomalies in the system are often difficult to capture, particularly during complex dynamic adjustments. Even if the final state of the system meets the expectations, it is challenging to determine whether any subtle errors or inconsistencies occurred during the adjustment process. Due to their subtlety and occasional nature, such anomalies are often difficult to detect. For these undetectable anomalies, we cannot predict whether they will have a minor or even latent long-term effect on the system.

Limitation in interaction and future work. Currently, we focus only on detecting and correcting the state of a single operator, without considering the complex interactions between multiple operators. In cloud-native environments, multiple operators manage resources, and their interactions can cause state conflicts, resource contention, or dependency issues. These anomalies often exceed the scope of single-operator mechanisms, making integrated detection and correction across operators a critical challenge. In future work, we plan to address interactions between multiple Kubernetes operators, aiming to enhance the robustness and reliability of cloud-native systems. Our goal is to provide a more comprehensive solution for Kubernetes operators in large-scale production environments.

VII. Conclusion

This paper introduces Arsc, a technique for generating, detecting, and correcting the state of Kubernetes clusters. The system state anomalies we detected can be categorized into three types: incorrect state, unhealthy state, and system crash. The causes of these anomalies include internal reconciliation issues and external disruptive errors. We propose three correction strategies to address these anomalies, depending on their underlying causes. The results show that Arsc demonstrates strong performance in maintaining state correctness in operator-based Kubernetes systems. Our future research goal is to address the state correctness issues in systems involving interactions between multiple operators, aiming to enhance the reliability and consistency of the system in more complex environments.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 62422208, 62202276, 62232010), in part by Shandong Science Fund for Excellent Young Scholars (No. 2022HWYQ-038), in part by Shandong Science Fund (No. 2023TSGC0105, 2024TSGC0228), in part by Ministry of Industry and Information Technology of China: Cloud-Data Integrated Machine for Cloud-Edge Collaboration.

References

[1] "Kubernetes." https://kubernetes.io/.
[2] "Swarm." https://docs.docker.com/engine/swarm/.
[3] "Apache mesos." https://mesos.apache.org/.
[4] "Operatorhub.io." https://operatorhub.io/.
[5] S. Engineering, "Fleet management at spotify (part 2): The path to declarative infrastructure." https://engineering.atspotify.com/2023/05/fleet-management-at-spotify-part-2-the-path-to-declarative-infrastructure/, 05 2023.
[6] Databricks, "Scaling apache spark on kubernetes at lyftli gao lyft,rohit menon lyft." https://www.youtube.com/watch?v=PPtrY_XxYBE.
[7] "Demystifying kubernetes as a service - how alibaba cloud manages 10,000s of kubernetes clusters." https://www.cncf.io/blog/2019/12/12/demystifying-kubernetes-as-a-service-how-does-alibaba-cloud-manage-10000s-of-kubernetes-clusters/, 12 2019.
[8] "Custom resources." https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.
[9] "Controller." https://kubernetes.io/docs/concepts/architecture/controller/.
[10] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 143–159, 2022.
[11] B. Liu, G. Lim, R. Beckett, and P. B. Godfrey, "Kivi: Verification for cluster management," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pp. 509–527, 2024.
[12] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic end-to-end testing for operation correctness of cloud system management," in *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 96–112, 2023.
[13] T. Hockin, "Edge vs. level triggered logic." https://speakerdeck.com/thockin/edge-vs-level-triggered-logic, 2017.
[14] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres, *et al.*, "Anvil: Verifying liveness of cluster management controllers," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 649–666, 2024.
[15] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Verifying concurrent, crash-safe systems with perennial," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 243–258, 2019.
[16] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, *et al.*, "Using lightweight formal methods to validate a key-value storage node in amazon s3," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 836–850, 2021.
[17] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno, "Storage systems are distributed systems (so verify them that way!)," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 99–115, 2020.
[18] N. Yaseen, B. Arzani, R. Beckett, S. Ciraci, and V. Liu, "Aragog: Scalable runtime verification of shardable networked systems," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 701–718, 2020.
[19] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen, "Using concurrent relational logic with helpers for verifying the atomfs file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 259–274, 2019.
[20] H. Zhang, W. Honoré, N. Koh, Y. Li, Y. Li, L.-Y. Xia, L. Beringer, W. Mansky, B. Pierce, and S. Zdancewic, "Verifying an http key-value server with interaction trees and vst," in *The 12th Conference on Interactive Theorem Proving*, p. 32, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
[21] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: a framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 357–368, 2015.
[22] R. Tao, J. Yao, X. Li, S.-W. Li, J. Nieh, and R. Gu, "Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 866–881, 2021.
[23] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang, "Nickel: A framework for design and verification of information flow control systems," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 287–305, 2018.
[24] V. Arun, M. T. Arashloo, A. Saeed, M. Alizadeh, and H. Balakrishnan, "Toward formally verifying congestion control behavior," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pp. 1–16, 2021.
[25] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 155–168, 2017.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.

[26] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "{NetSMC}: A custom symbolic model checker for stateful network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 181–200, 2020.
[27] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness verification of stateful network functions," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 257–272, 2020.
[28] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 59–72, 2015.
[29] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak, "Predicting and preventing inconsistencies in deployed distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 1, pp. 1–49, 2010.
[30] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 1–17, 2015.
[31] G. Turin, A. Borgarelli, S. Donetti, E. B. Johnsen, S. L. Tapia Tarifa, and F. Damiani, "A formal model of the kubernetes container framework," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 558–577, Springer, 2020.
[32] B. Liu, A. Kheradmand, M. Caesar, and P. B. Godfrey, "Towards verified self-driving infrastructure," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pp. 96–102, 2020.
[33] H. Ding, Z. Wang, Z. Shen, R. Chen, and H. Chen, "Automated verification of idempotence for stateful serverless applications," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 887–910, 2023.
[34] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 174–186, 1997.
[35] J. Yang, C. Sar, and D. Engler, "Explode: a lightweight, general system for finding serious storage system errors," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 131–146, 2006.
[36] "How an operator becomes the hero of the edge." https://www.kubermatic.com/resources/how-an-operator-becomes-the-hero-of-the-edge/, 2021.
[37] C. Hall, "Aws, google, microsoft, red hat's new registry to act as clearing house for kubernetes operators." https://www.datacenterknowledge.com/open-source/aws-google-microsoft-red-hats-new-registry-act-clearing-house-kubernetes-operators, 03 2019.
[38] "Why operators are essential for kubernetes." https://www.redhat.com/en/blog/why-operators-are-essential-kubernetes, 2021.
[39] "Introducing the aws controllers for kubernetes (ack) — amazon web services." https://aws.amazon.com/cn/blogs/containers/aws-controllers-for-kubernetes-ack/, 08 2020.
[40] "Artifacthub." https://artifacthub.io/.
[41] "Official cve feed." https://kubernetes.io/docs/reference/issues-security/official-cve-feed/.
[42] "kind." https://kind.sigs.k8s.io/.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on October 22,2025 at 04:45:53 UTC from IEEE Xplore. Restrictions apply.