Anvil: Verifying Liveness of Cluster Management Controllers

Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, and Zicheng Ma, *University of Illinois Urbana-Champaign;* Tej Chajed, *University of Wisconsin-Madison;* Jon Howell, Andrea Lattuada, and Oded Padon, *VMware Research;* Lalith Suresh, *Feldera;* Adriana Szekeres, *VMware Research;* Tianyin Xu, *University of Illinois Urbana-Champaign*

https://www.usenix.org/conference/osdi24/presentation/sun-xudong

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Anvil: Verifying Liveness of Cluster Management Controllers

Xudong Sun[†], Wenjie Ma[†], Jiawei Tyler Gu[†], Zicheng Ma[†], Tej Chajed[‡],
Jon Howell[◇], Andrea Lattuada[◇], Oded Padon[◇], Lalith Suresh[⋆], Adriana Szekeres[◇], Tianyin Xu[†]

[†]University of Illinois Urbana-Champaign [‡]University of Wisconsin-Madison
[◇]VMware Research [⋆]Feldera

Abstract

Modern clouds depend crucially on an extensible ecosystem of thousands of controllers, each managing critical systems (e.g., a ZooKeeper cluster). A controller continuously *reconciles* the current state of the system to a desired state according to a declarative description. However, controllers have bugs that make them never achieve the desired state, due to concurrency, asynchrony, and failures; there are cases where after an inopportune failure, a controller can make no further progress. Formal verification is promising for avoiding bugs in distributed systems, but most work so far focused on safety, whereas reconciliation is fundamentally not a safety property.

This paper develops the first tool to apply formal verification to the problem of controller correctness, with a general specification we call *eventually stable reconciliation*, written as a concise temporal logic liveness property. We present Anvil, a framework for developing controller implementations in Rust and verifying that the controllers correctly implement eventually stable reconciliation. We use Anvil to verify three Kubernetes controllers for managing ZooKeeper, RabbitMQ, and FluentBit, which can readily be deployed in Kubernetes platforms and are comparable in terms of features and performance to widely used unverified controllers.

1 Introduction

Modern clouds are powered by cluster managers such as Kubernetes [12], Borg [89], ECS [73] and Twine [87]. These systems manage large-scale cluster resources and all applications running atop them. Architecturally, these systems comprise a collection of *controllers* that implement all the cluster-management logic based on the *state reconciliation* principle [4, 30]. Controllers are loosely coupled microservices, each monitoring the cluster state and continuously reconciling the *current* cluster state to match a *desired* state. In Kubernetes for example, controllers manage everything from system resources (e.g., pods, data volumes, networking, and stateful services) to application lifecycles (e.g., provisioning, upgrades, and scaling). There is a thriving ecosystem of thousands of domain-specific controllers that extend Kubernetes [46, 47, 60, 81, 84]. All these controllers perform critical operations, making their correctness paramount.

Implementing correct controllers is immensely challenging, due to the enormity of cluster state space and the complexity of failure events (e.g., node crashes, network interruptions, and asynchrony issues). Recent automated controller testing tools [44, 85] found many bugs with severe consequences such as system outages, data loss, and resource leaks in popular Kubernetes controllers. Buggy controllers have caused many production incidents [45, 57, 71, 74].

This paper addresses the controller correctness challenge with two major contributions: (1) *eventually stable reconciliation*, a general specification for controller correctness which we develop as a liveness property, and (2) Anvil, a framework for implementing practical controllers and formally verifying that a controller implements eventually stable reconciliation. We have developed and verified practical Kubernetes controllers for managing critical systems using Anvil.

Challenges and contributions. Addressing controller correctness with formal verification poses several challenges. The first challenge is to define a correctness specification that is generally applicable to diverse controllers, powerful enough to preclude a broad range of bugs, and concise enough for manual inspection, together with appropriate assumptions that make it possible to implement the specification. The second challenge lies in proving that the controller implements this specification: controllers are complex, feature-rich real-world systems that do not have pen-and-paper proofs that we can reference. This problem is exacerbated by the fact that controllers run in a complex and dynamic environment, where the controller must handle unexpected faults, asynchrony, and conflicts when interacting with other controllers.

We present Eventually Stable Reconciliation (ESR) as a general specification of controller correctness (§3). ESR is a liveness property, which states that a controller should *eventually* reconcile the cluster to a desired state, and then *always* keep the cluster in the desired state. ESR captures the essential functionality that controllers should provide in a precise language, and it precludes a broad range of bugs caused by factors like inopportune failures and conflicts with other controllers. ESR is also realistic and captures the necessary premise to reach the desired state. We formalize ESR as a concise Temporal Logic of Actions (TLA) [58] formula.

A common challenge in proving liveness is that the proof depends on subtle *fairness assumptions*, including assumptions about possible faults. Overly strong assumptions (e.g., the controller can crash at most once) lead to weak correctness guarantees, and overly weak assumptions (e.g., the controller can keep crashing forever) make liveness verification untenable. Anvil employs an assumption that covers a broad range of fault scenarios—an arbitrary number of faults can happen, but eventually faults stop happening. This assumption is similar in spirit to partial synchrony [40] but for faults.

To prove that a controller satisfies ESR, one must consider the controller's interactions with the environment in which it runs. Anvil models this environment, including the shared cluster state, asynchronous network, other controllers, and a realistic fault model (§4.3). The environment model also encodes assumptions on fair scheduling and faults. Anvil abstracts general liveness reasoning patterns in the environment into reusable lemmas to reduce proof effort (§4.4).

With the reusable models and lemmas provided by Anvil, developers can prove that the controller makes progress from any cluster state towards potential desired states in the presence of asynchrony, faults, and conflicts with other controllers. We present a proof strategy to disentangle the challenges of proving ESR (§5), which divides the proof into two lemmas: (1) starting from any possible state resulting from potential interleaving of previous execution and faults, the controller progresses towards the desired state in a stable environment, and (2) the environment eventually becomes stable. Both lemmas can be proven using the temporal proof rules that Anvil provides (under the fairness assumptions). We have applied this proof strategy to verify three controllers using Anvil.

Implementation. We implemented Anvil for verifying Kubernetes controllers on top of Verus [61], an SMT-based deductive verification tool for Rust. With Verus, developers can implement controllers in Rust and formally verify their implementations. Verus does not support temporal logic reasoning, so Anvil provides a TLA embedding on top of first-order logic (§4.2) to enable TLA-style temporal reasoning.

We used Anvil to implement three practical Kubernetes controllers for managing ZooKeeper, RabbitMQ, and FluentBit (§6). These controllers can readily be deployed in real-world Kubernetes platforms; they provide feature parity and competitive performance w.r.t. existing mature, widely used (but unverified) controllers. The verification effort is manageable, with the proof-to-code ratio ranging from 4.5 to 7.4 across the controllers. The verification process exposed deep bugs in both our early implementations and unverified reference controllers. Although Anvil is primarily designed for liveness verification, it also supports safety verification; we prove a safety property specific to the RabbitMQ controller: the controller *never* performs unsafe scaling operations.

Summary. This paper makes the following contributions:

- Eventually stable reconciliation (ESR), a general specification for controller correctness as a liveness property;
- Anvil, a framework for developing practical controllers and formally verifying that the controller implementations satisfy correctness properties such as ESR;
- three representative and practical Kubernetes controllers verified using Anvil; and
- an evaluation of the end-to-end correctness and performance of the three verified controllers.

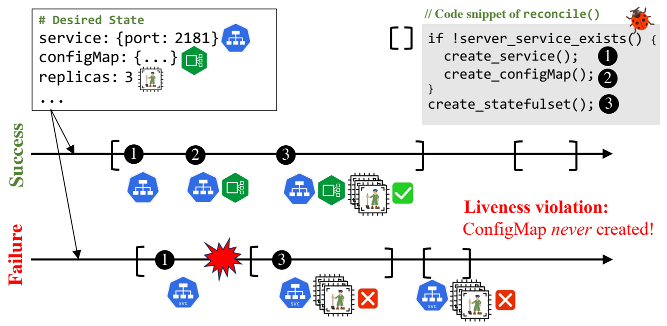
Anvil and the verified controllers are available at `https://github.com/vmware-research/verifiable-controllers`.

2 Implementing Controllers

Controllers follow the *state reconciliation* principle: each controller runs a control loop that continuously reconciles the cluster's current state to the desired state [4, 8]. At each loop iteration, a *reconciliation* procedure checks whether the current cluster state matches the desired state; if not, it performs corrective operations to move the cluster towards the desired state (e.g., launching new replicas in an ensemble of servers when existing replicas fail). The operations query or update the cluster state, represented by shared data objects. These state objects are exposed by REST-based API servers and are stored in a logically centralized data store like etcd [1]. The desired cluster state is described declaratively and can be dynamically updated during the lifecycle of a running controller. The reconciliation procedure is typically implemented in a `reconcile()` function, which is invoked whenever the desired state description (or its relevant cluster states) is changed.

Figure 1 exemplifies the reconciliation process of a Kubernetes controller for managing ZooKeeper. To create a ZooKeeper cluster, the controller takes three steps to create: ❶ a networked service (a Kubernetes service object [17]), ❷ a ZooKeeper configuration (a config map object [14]), and ❸ a stateful application (a stateful set object [18]) with three replicas. Each step is performed by creating a new state object of the corresponding resources via the Kubernetes API, which then triggers Kubernetes built-in controllers, e.g., the StatefulSet controller will create three sets of pods and volumes to run containerized ZooKeeper nodes. In the end, the cluster state matches the desired state. Later, if the desired state changes (e.g., its `replicas` is increased), the ZooKeeper controller will start a new iteration of reconciliation that updates the stateful set object to scale up the ZooKeeper cluster.

Correctness challenges. A bug in a controller's reconciliation can result in the controller *never* being able to match the desired state, even when `reconcile()` is called repeatedly. Controllers are expected to be level-triggered [53]: `reconcile()` can be called from any current cluster state to match any given desired state, with no guarantee that the controller has seen the entire history of cluster state changes [86]. In addition, controllers must tolerate unexpected failures and asynchrony while running `reconcile()`, which leads to a state-space explosion that makes testing controllers difficult. Figure 1 shows one of many bug patterns



```
# Desired State
service: {port: 2181}
configMap: {...}
replicas: 3
...
```

```
// Code snippet of reconcile()
if !server_service_exists() {
  create_service();     ❶
  create_configMap();   ❷
}
create_statefulset();   ❸
```

Figure 1: **An example of state reconciliation of an unverified ZooKeeper controller.** A liveness bug is triggered by a crash during reconciliation. The bug pattern is common in real-world Kubernetes controllers (known as intermediate-state bugs [85]).

```
pub trait Controller {
  type D; // desired cluster state description
  type S; // local state in the state machine

  /// Returns the initial local state (in the state
  /// machine) of every reconcile()
  fn initial_state() -> S;

  /// Returns S: next local state in state machine
  /// Req: external request (e.g. to Kubernetes)
  /// # Arguments
  /// * d: the desired cluster-state description
  /// * r: response to the request from last step
  /// * s: current local state in state machine
  fn step(d: &D, r: Resp, s: S) -> (S, Req);

  /// Returns true if all steps are done
  fn done(s: &S) -> bool;

  /// Returns true for error states
  fn error(s: &S) -> bool;
} // other advanced APIs are omitted
```

Figure 2: **Anvil's basic Controller API.** To implement a controller, developers implement the Controller trait.

of controllers [35, 44, 54, 64, 85]. If the controller crashes between steps ❶ and ❷ during an execution of `reconcile()`, Kubernetes will reboot the controller. The freshly invoked `reconcile()` call now faces the intermediate state created by the previous failed execution (❶). However, in this case, the controller would *never* perform ❷ due to a buggy predicate, which only checks whether the networked service exists, but not whether the config map also exists. As a result, the cluster state would never match the desired state—a liveness violation. Such liveness violations are notoriously hard to detect by testing or model checking [55].

Implementing controllers with Anvil. In Anvil, developers implement a controller using a state machine; this style is common practice in unverified controllers as well [2, 6], and in Anvil it enables TLA-style verification. Figure 2 shows a snippet of the Anvil Controller API specified using a Rust trait: it involves defining the initial state and the transitions of a state machine. Anvil's `reconcile()` uses the state machine as shown in Figure 3: it starts from the initial state and invokes `step()` iteratively until all steps are done or if any step encounters an error. Each iteration of `step()` returns the next

```
pub fn reconcile<C>(d: C::D) -> Result<Action, Error>
  where C: Controller {
  let mut s = C::initial_state();
  let mut resp = None;
  loop { // exercise the state machine
    if C::error(&s) {
      return Err(ErrorNeedsRequeue);
    } else if C::done(&s) {
      return Ok(requeue(timeout));
    }
    let (next_s, req) = C::step(&d, resp, s);
    resp = send_external_request::<C>(req);
    s = next_s;
  }
} // details like validity checks are omitted
```

Figure 3: **Anvil code that assembles `reconcile()` using the Controller API in Figure 2.**

```
fn step(d: &ZKD, r: Resp, s: ZKS) -> (ZKS, Req) {
  match s {
    CheckService => { // if the service exists
      let service_get_req = KubeGet { ... }
      return (ReconcileService, service_get_req);
    }
    ReconcileService => {
      /// create/update the service based on response r
      if r.is_ok() {
        let service_update_req = ...;
        return (CheckConfigMap, service_update_req);
      } else if r.is_not_found() {
        let service_create_req = ...;
        return (CheckConfigMap, service_create_req);
      } else {
        return (Error, Noop); // restart reconcile()
      }
    }
    CheckConfigMap => { ... }
    ReconcileConfigMap => { ... }
    CheckStatefulSet => { ... }
    ReconcileStatefulSet => { ... }
    ...
  } // more step branches are omitted
}
```

Figure 4: **A simplified implementation of `step()` using Anvil for creating a ZooKeeper cluster.** Proof-related code is omitted.

state in the state machine, together with an external request. The external request is typically a REST call to Kubernetes APIs, but can also be extended to non-Kubernetes APIs (§6.1). The response to the external request is passed as an argument to the next iteration of `step()`. Note that the API enforces no more than one external request per `step()`, making the state-machine transition atomic with respect to cluster-state changes. Anvil's `reconcile()` interfaces a trusted Kubernetes client library (kube-rs [11]) which invokes `reconcile()` upon changes, handles its output, and requeues the next invocation.

Figure 4 shows the `step()` implementation of a ZooKeeper controller (Figure 1). The `step()` function takes the desired state description of the ZooKeeper cluster (`d`), the response (`r`) to the request from last step (if any), and the current local state (`s`), and deterministically returns the next local state and the external request. The state machine starts from the `CheckService` state, where it returns a request to read the service object [17] from the Kubernetes API (`service_get_req`) and the next state to transition to `ReconcileService`. The `reconcile` method (Figure 3) fetches the service object using the Kubernetes API, and moves on to the next iteration of `step()`, bringing the state machine to `ReconcileService` branch. The controller proceeds to create or update the service based on the response of `service_get_req`. In this way, the controller progressively reconciles each cluster-state object and eventually matches the desired state declared by `ZKD`.

Next, we present a general controller correctness specification, termed eventually stable reconciliation, in §3, and then explain how to verify it using Anvil in §4 and §5.

3 Eventually Stable Reconciliation (ESR)

Controllers reconcile the cluster state to match the desired state. While the details vary between controllers, and some controllers may have additional correctness guarantees, we formalize a general property called *eventually stable reconciliation (ESR)* that captures this ubiquitous pattern.

ESR captures two key properties of any controller's state reconciliation behavior: (1) *progress*: given a desired state description, the controller must eventually make the cluster state match that desired state (unless the desired state changes), and (2) *stability*: if the controller successfully brought the cluster to the desired state, it must keep the cluster in that state (unless the desired state changes).

To make our specification general, we do not wish to commit to any particular bound on the time or number of operations the controller takes to bring the cluster to the desired state. So, we want to talk about guaranteed *eventualities*. Such unbounded eventualities are naturally described using *temporal logics* [80]. We use TLA (temporal logic of actions) [58], a linear-time temporal logic well-suited to our needs. TLA is designed for reasoning about a system described as a state machine. The behavior of the state machine is captured by its set of traces, infinite sequences of system states where the first state is a valid initial state and each subsequent state is obtained via a valid transition from the previous state.

We formalize ESR as a TLA formula that should hold for *all* traces of the system's execution, where the system includes both the controller and its environment, under all possibilities for asynchrony, concurrency, and faults (e.g., controller crashes). We use d to denote a state description, $\texttt{desire}(d)$ to denote whether d is the current description of the desired state, $\texttt{match}(d)$ to denote whether the current cluster state matches the description d. Our definition of ESR is given by the following formula:

$$\forall d. \Box\big(\Box\texttt{desire}(d) \Rightarrow \Diamond\Box\texttt{match}(d)\big). \tag{1}$$

Informally, ESR asserts that if at some point the desired state stops changing, then the cluster will eventually reach a state that matches it, and stay that way forever. The temporal operators $\Diamond$ (eventually) and $\Box$ (always) are used in temporal logics to reason about the future of an execution trace. If a predicate P talks about the current state, then $\Box P$ says that P holds in the current state and all future states, while $\Diamond P$ says that P holds in the current or some future state. Temporal logics such as TLA also allow nesting of temporal operators; for example, $\Diamond\Box P$ means that eventually we get to a point such that from that point onwards, P always holds.

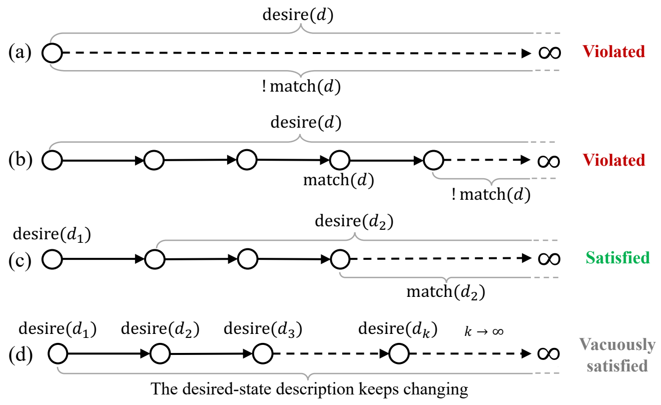


Figure 5: **Executions that violate or satisfy ESR.**

The formalization of ESR is a key contribution of this paper in that it captures the key correctness properties shared by virtually all controllers: progress and stability. We elaborate on this with a detailed dissection of eq. (1).

The innermost conclusion of the formula is $\Diamond\Box\texttt{match}(d)$, which states that eventually ($\Diamond$) the controller matches the desired state (progress), and from then on, it always ($\Box$) keeps the cluster state at the desired state (stability). In front of this expression, $\Box\texttt{desire}(d)$ is a realistic and necessary premise for the controller to match the desired state—if the desired state description keeps changing forever, the controller will keep chasing a moving target forever, and nothing can be guaranteed as we do not wish to assume a bound on how long state reconciliation takes. The outer $\Box$ in eq. (1) says that $\Box\texttt{desire}(d) \Rightarrow \Diamond\Box\texttt{match}(d)$ always holds, meaning that the controller continuously reconciles the cluster state *regardless of its past execution*. Finally, the $\forall d$ states that the controller reconciles all desired state descriptions.

Figure 5 illustrates the ESR definition in some examples, some that satisfy the definition and others that do not: (a) violates progress because the cluster state never matches d, (b) violates stability because the cluster state first matches d but then deviates from d, (c) satisfies ESR because the cluster state eventually matches and always matches d_2, and (d) vacuously satisfies ESR because the desired state never stops changing, so $\Box\texttt{desire}(d)$ does not hold for any fixed d.

The verification goal for each controller is to prove that the controller satisfies ESR—*all* possible executions of the controller satisfy ESR. We use `model` to describe all possible executions of the controller that runs in an environment with asynchrony, concurrency and faults. We use $\leadsto$ (leads-to) notation to simplify the presentation of the ESR property, where $P \leadsto Q$ means $\Box(P \Rightarrow \Diamond Q)$. Then the statement that the controller satisfies ESR is formalized as:

$$\texttt{model} \models \forall d. \Box\texttt{desire}(d) \leadsto \Box\texttt{match}(d). \tag{2}$$

The power of ESR. Strictly speaking, ESR (eq. (1)) only guarantees one successful state reconciliation—the one that happens after the desired state stops changing forever. However, in practice the controller has no way of knowing if the desired state will change in the future or not. Therefore, we can expect that a controller that satisfies ESR will bring the cluster to match the desired state (and keep it like that) for any desired state that remains unchanged for *long enough*. ESR achieves this without getting into the gory details of defining exactly how long is long enough. Note further that because of the outermost $\Box$ in eq. (1), a controller that satisfies ESR will deliver *multiple* successful state reconciliations, assuming that the desired state goes through a series of slow changes.

Our analysis shows that ESR can ensure the absence of a broad range of controller bugs [44, 64, 85]. For example, recent testing tools [44,85] detected 70 bugs across 16 popular controllers that the controller never matches the desired state due to improper handling of corner-case state descriptions, inopportune failures and concurrency issues, which consist of 69% of all the detected bugs. All such bugs are precluded by ESR. Prior work [64] also reported failure patterns where the cluster state, after matching a desired state, then deviates due to conflicting interactions with other controllers. Such bugs, as stability violations, are also precluded by ESR.

4 Anvil

Anvil is a framework for developing controllers and mechanically proving that they implement correctness specifications like ESR. Anvil is built on top of Verus [61], an SMT-based deductive verification tool for Rust backed by Z3 [39], in similar spirit to Dafny [62]; it offers a Hoare-logic [52] framework for reasoning modularly about imperative code in Rust.

Figure 6 shows the workflow of using Anvil to verify a controller. The developer first provides Ⓐ a *controller model* (an abstract state machine) and then proves two theorems: Ⓑ the Controller trait implementation (Figure 4) conforms to the controller model and Ⓒ the controller model, together with a model of the environment (e.g., the network, other controllers, faults), satisfies specifications like ESR (eq. (2)).

Writing the controller model and verifying the implementation conforms to the model are straightforward. The controller model is an abstract state machine with the same structure as the implementation state machine. To prove conformance, developers prove that each step in the implementation corresponds to exactly one step in the model using standard Floyd-Hoare style reasoning (§4.1). Note: the controller model is written in Verus' specification language to enable verification.

Verifying the model entails ESR is more challenging: developers need to apply temporal logic reasoning on the interaction between the controller and its environment (including faults) at the model level to prove ESR. To reduce developers' burden on specification and proof, Anvil provides (1) a TLA embedding (§4.2) that defines temporal logic operators on top of first-order logic to enable specification and proof in temporal logic (Verus does not support temporal logic), (2) a model of the controller environment (§4.3), including components that a controller interacts with, faults that a controller must tolerate, and reasonable assumptions on fair scheduling and faults that controller liveness depends on, and (3) reusable lemmas (§4.4) that encode temporal proof rules and liveness and safety properties of the interactions between a controller and the environment; these lemmas can be directly assembled into developers' ESR proofs.

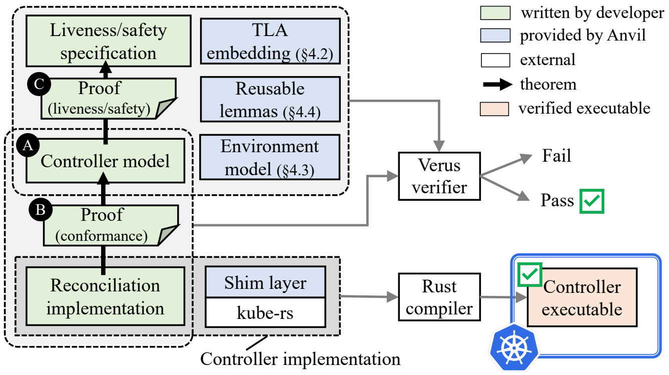


Figure 6: **An overview of Anvil's workflow.**

In this section, we explain how Anvil supports the verification of controllers and then present a general, effective strategy for developing proofs to verify ESR in §5.

Assumptions. Anvil relies on the following assumptions: (1) The TLA embedding correctly defines TLA concepts [58]. (2) The controller environment model correctly describes the interactions between the controller and its environment. (3) The specification of the unverified APIs for querying and updating the cluster state correctly describes the behavior of these APIs. (4) The verifier (Verus and Z3), the Rust compiler, and the underlying operating system are correct.

4.1 Controller Model

To verify controller correctness, developers first write a controller model and prove the controller implementation conforms to this model, similar to prior work [49, 51]. The controller model is a mathematical, state-machine representation of the imperative controller implementation, which abstracts the data types in the implementation and enables TLA-style verification. Given the proof of implementation-model conformance, the model is not assumed to be correct in Anvil's overall verification guarantee.

Anvil provides an API for developers to write the controller model, shown in Figure 7. This API defines a state machine and is similar to the Controller API in Figure 2, except that all the methods and variables are written in *ghost code* [61, 62]. Ghost code is auxiliary code that describes properties of programs and is used for verification only—the code is erased before compilation and thus poses no runtime overhead. Concretely, in the controller model, all the methods are Verus' `spec` functions which are purely functional, and all the variables are ghost types that represent an abstract view of the variables in the implementation, e.g., a heap-allocated Rust `Vec` is represented as a mathematical sequence (Verus' `Seq`).

```
pub trait ControllerModel {
  type DV; // view of the desired state description
  type SV; // view of local state in the state machine
  spec fn m_initial_state() -> SV;
  spec fn m_step(d: DV, r: RespV, s: SV) -> (SV, ReqV);
  spec fn m_done(s: SV) -> bool;
  spec fn m_error(s: SV) -> bool;
} // other advanced APIs are omitted
```

Figure 7: **Anvil's ControllerModel API.** Developers use the API to write the controller model (an abstract state machine). It mirrors the implementation trait (Figure 2) but is written in ghost code.

Given a controller implementation, writing the controller model is straightforward. Given a `step()` implementation in the Controller API (Figure 2), developers write a corresponding `m_step()` using the ControllerModel API (Figure 7). If the implementation's `step()` returns a Kubernetes-API request, the model's `m_step()` correspondingly returns a ghost-type request (`ReqV`) that queries the Kubernetes API model (§4.3.1). The other trait methods are largely identical to their counterparts in the implementation except for the data types.

For each implementation data type defined by developers, such as the types for the desired state description and the state machine's local state (e.g., `D` and `S` in Figure 2), developers need to define a corresponding ghost type (e.g., `DV` and `SV`), typically by replacing implementation data types with corresponding ghost types. For example, if `D` has a field of Rust `Vec` type, `DV` will have a field of Verus `Seq` type. Developers also need to define a `view()` function that converts an implementation object to the corresponding ghost-type object.

Implementation-model conformance. Developers need to prove that the implementation state machine has the same initial state, transitions and termination conditions as the model state machine through `view()`. Figure 8 shows the theorem to prove conformance for the ZooKeeper controller's `step()` in Figure 4. This theorem states that the model's `m_step()` produces the same output (in ghost types), given the same input (in ghost types) of the implementation's `step()`.

```
fn step(d: &ZKD, r: Resp, s: ZKS) -> (res: (ZKS, Req))
  ensures res@ == ZKControllerModel::m_step(d@, r@, s@)
{ ... } // implementation body is omitted
```

Figure 8: **The conformance theorem written as a postcondition of `step`.** The `step` function is executable (part of the controller implementation). The symbol `@` is a shorthand for `.view()` in Verus, which converts an implementation type into a ghost type.

The key challenge in enabling and automating the conformance proof is to reason about data types defined in external, unverified libraries. For example, the controller implementation needs to use data types that define Kubernetes state objects from the kube-rs [11] library, but Verus cannot directly reason about definitions from unverified libraries. So, Anvil defines wrappers that translate every Kubernetes state-object type to its corresponding ghost type. These wrappers are straightforward to implement and are trusted; Anvil includes unit tests that cover *all* the trusted wrapper methods.

The controller implementation uses the wrapper types instead of raw types from kube-rs, and the model uses the corresponding ghost types. For verification, Verus automatically tracks the wrapper's view (`view()`) through the postconditions of the wrapper methods used in the controller implementation. Verus compares the object's view to the ghost object used in the controller model to check the conformance proof; e.g., to prove the theorem in Figure 8, Verus compares the returned request's view and its counterpart in the model.

With this design, the conformance proof is done by standard Floyd-Hoare style reasoning [52] and is largely automated by Verus. Most of the manual proof effort is the requirement to ask Verus to prove two objects are equal if they have the same properties, e.g., to prove a `Vec`'s view (in the implementation) and the corresponding `Seq` (in the model) are equal.

4.2 TLA Embedding

To enable liveness reasoning on top of Verus, Anvil develops a TLA embedding that models important concepts in TLA. Anvil follows IronFleet [51] and models three major concepts as follows: (1) *an execution* is an infinite sequence of system states encoded as a mapping from natural numbers to states, (2) *a temporal predicate* is a boolean predicate on executions, and (3) *a temporal operator* (e.g., $\Diamond$, $\Box$ and $\leadsto$) is a function that transforms one temporal predicate into another. Every temporal operator is defined using only first-order quantifiers on executions. Suppose `P` is a temporal predicate and `ex` is an execution, `eventually(P)` (resp. `always(P)`) is a temporal predicate that holds true of `ex` if `P` is true on some (resp. all) suffixes of `ex`, that is, at some (resp. all) future time.

With the TLA embedding, developers can specify the theorem that the controller satisfies ESR (eq. (2)) as in Figure 9. The definition of `desire` is typically reused among controllers but can also be extended if more premises are required for liveness. The definition of `match` varies across controllers; e.g., the `match(d)` for the ZooKeeper controller in Figure 4 checks if the service, config map and stateful set exist in the data store and match the desired state description `d` (Figure 10).

```
// model ⊨ ∀d.□desire(d) ⇝ □match(d)
model.entails(
  forall(|d: DV|
    always(desire(d)).leads_to(always(match(d)))
  ))
```

Figure 9: **The ESR theorem specified using the TLA embedding.**

In the style of specifying systems [59], Anvil diligently abstracts away executions: developers model components at the levels of state and action (transition between states), then complete liveness proofs with temporal operators. Essentially, Anvil encourages developers to express concepts as *state*

```
spec fn match(d: ZKDV) -> TemporalPredicate {
  lift(|s: ClusterState| { // lift a state predicate
    let store = s.state_object_data_store;
    store.contains(service_name(d))
    && store.contains(config_map_name(d))
    && store.contains(stateful_set_name(d))
    && store[stateful_set_name(d)].replicas == d.size
    && ... // more statements are omitted
  })
}
```

Figure 10: **The definition of the ZooKeeper controller's `match`.** The temporal predicate, when applied to an execution, checks the first state to see if the state objects exist in `store` and match `d`. `ZKDV` is the view of the ZooKeeper desired state description (`ZKD`).

predicates over individual states or *action predicates* over individual transitions. Developers can convert a state predicate to a temporal predicate using a `lift` function [59]: an execution satisfies the lifted predicate if its first state satisfies the state predicate; lifting an action predicate likewise applies the predicate to the first two states of an execution. For example, the temporal predicate `match(d)` is defined by lifting a state predicate as shown in Figure 10. In this way, developers focus on reasoning about individual states and actions when proving invariants and lift them to temporal predicates when applying temporal proof rules (§4.4.1). This differs from IronFleet which interacts directly with instantiated executions throughout the liveness proof. We present Anvil's temporal reasoning style in detail in §5.2.

4.3 Modeling Controller Environment

To reason about interactions between a controller and its environment, Anvil models the controller environment. The goal is to describe the external behavior of different components in the environment and capture the factors that affect a controller's correctness, including asynchrony, concurrency and faults. To this end, Anvil models the environment as a compound state machine, consisting of individual state machines that depict the behavior of different components, such as the network and the API server, as well as faults. The environment model also comes with reasonable assumptions on fair scheduling and faults that liveness depends on.

4.3.1 Modeling Environment Components

Anvil models the environment as a compound state machine with each inner individual state machine modeling one component that a controller interacts with, including:

- an asynchronous network that delivers messages among components with no ordering guarantees;
- the cluster-state data store and the API server; the cluster state is stored in the logically centralized data store (e.g., etcd [1]) and exposed by the API server which handles the controller's query or update requests;
- other controllers in the environment that might interact with the to-be-verified controller; and
- clients that request desired cluster states; clients can update the desired cluster state at any time.

Anvil embeds the controller model in the compound state machine to reason through the interaction between the controller and its environment. The compound state machine, in each step, chooses one individual state machine and invokes one step of that state machine. All the steps are atomic regarding how the cluster state advances (e.g., the API server only handles one request to update the cluster state in each step).

The compound state machine model naturally captures asynchrony and concurrency challenges for controllers. For example, time-of-check to time-of-use (TOCTOU) issues can happen when the cluster state has changed since the last time the controller queried it, but the controller issues an update based on its stale view of the cluster state.

A model of Kubernetes environment. Anvil models the Kubernetes cluster-state data store as a map that stores state objects. Anvil models Kubernetes API servers' mechanisms for validating and coordinating controller requests, including its multi-version concurrency control mechanism wherein each object is versioned. Requests from the controllers must be validated with a version check to take effect.

Anvil models Kubernetes built-in controllers that interact with other controllers, including (1) the garbage collector [16] which deletes a state object if all of its listed owners have been deleted, (2) the StatefulSet controller [18] which manages stateful applications, and (3) the DaemonSet controller [15] which manages daemons (e.g., for monitoring) on every node.

4.3.2 Modeling Faults

Anvil models common faults that happen in modern clusters as actions in the compound state machine; the compound state machine in each step chooses to either let one component take one step or let one fault happen. Anvil models two types of faults: (1) *controller crash*: the controller can crash and reboot an arbitrary number of times. Each crash makes the controller stay offline for an arbitrary number of steps before it is rebooted. After a crash, the controller loses its internal (in-memory) state and has to start over from the beginning of its reconciliation procedure. (2) *request failures*: any request sent by the controller can fail at any point due to network timeouts or the API server being busy.

4.3.3 Specifying Liveness Assumptions

Liveness verification needs careful assumptions. In a concurrent, asynchronous system, *fairness assumptions* are needed to prove that something eventually happens as it relies on the system and its environment getting a chance to take certain actions—a property that is expected to hold in practice but must be nonetheless explicitly incorporated in our formal assumptions. This problem is especially pronounced for controller liveness: a controller's reconciliation (1) relies on other components' actions to complete, and (2) can be interrupted by faults or conflicting actions from other controllers. Anvil makes assumptions that the environment eventually allows the controller to make progress.

Weak fairness assumptions on actions. Applying the *weak fairness* [58] assumption is effective to make the liveness property hold, without assuming any specific fair scheduling. A weak fairness assumption states that if an action A remains "enabled" (i.e., the action can possibly occur), the action eventually occurs: $\Box \texttt{enabled}(A) \leadsto A$. The predicate $\texttt{enabled}(A)$ is true, if for S (the first state of the execution), there *exists* a next state S' such that $A(S,S')$ is true; that is, it is possible for A to occur and transition to S'.

We include fairness assumptions in the model by assuming weak fairness on the actions of the controller and other components in the environment.

Assumptions on faults. Controller liveness also needs assumptions on faults. If the compound state machine chooses to reboot the controller in every step, the controller will never get a chance to finish reconciliation. However, overly strong assumptions like "the controller crashes only once" lead to weak correctness guarantees. To strike a balance, we assume that faults can happen an arbitrary number of times but eventually stop happening, in the spirit of partial synchrony [40].

To incorporate this assumption, we add a "disable-fault" action for each type of fault to the compound state machine. We then add the weak fairness assumption to disable-fault actions. That is, the disable-fault action eventually happens, after which the corresponding type of fault no longer happens.

Assumptions on other controllers. Controllers share the cluster state and thus can conflict with each other. A controller's liveness relies on conflicts being eventually resolved, which mandates assumptions on other controllers. In Kubernetes as an example, the built-in StatefulSet controller can compete with the target controller forever. Suppose the controller uses a stateful set to manage a stateful application and updates the stateful set to match the desired state description. At the same time, the StatefulSet controller continuously updates the stateful set to publish the current status of each running node. When the two controllers are updating the same object concurrently, only one can succeed [13]. Thus, the environment model can adversarially keep letting the target controller lose the race and never reach the desired state.

Anvil assumes that the StatefulSet controller eventually stops updating the stateful set *until* the target controller updates the stateful set again. Similar to the fault assumption, we add to our model an action (with weak fairness) that disables the built-in StatefulSet controller's updates on a stateful set; the target controller's successful update to this stateful set will enable the StatefulSet controller again. Anvil makes the same assumption on how the built-in DaemonSet controller updates daemon sets.

4.4 Reusable Lemmas

Proving ESR requires applying temporal proof rules to reason about the controller's interaction with the environment. This is challenging in two ways: (1) temporal reasoning does not have good automation because SMT solvers like Z3 lack decision procedures for temporal operators, and (2) the interaction between the controller and the environment is complex and is subject to asynchrony and faults. To reduce developers' proof effort, Anvil provides a library of reusable lemmas that encode (1) commonly used temporal proof rules and (2) generic reasoning patterns in the controller environment.

```
proof fn leads_to_transitive(
  model, P, Q, R: TemporalPredicate
)
  requires
    model.entails(P.leads_to(Q)),
    model.entails(Q.leads_to(R))
  ensures model.entails(P.leads_to(R))
{ ... } // proof body is omitted
```

Figure 11: **The leads-to transitivity lemma.**

4.4.1 Temporal Reasoning Lemmas

Anvil provides temporal reasoning lemmas that encode commonly used proof rules to improve temporal reasoning automation. These lemmas are useful for proving liveness for any controller. One example is the *leads-to transitivity* lemma (Figure 11). It shows that if $P \leadsto Q$ and $Q \leadsto R$, then $P \leadsto R$, all under the same assumption *model*. The proof of this lemma involves using the temporal logic definitions, reasoning about an arbitrary time in an execution where P holds, and showing there exists a corresponding time where R eventually holds (using an intermediate time when Q holds, as guaranteed by the preconditions). In return, the developer can easily invoke the lemma without reference to execution or specific indices (these are hidden in the temporal logic lemmas). The leads-to transitivity lemma is frequently used for chaining leads-to formulas to deduce ESR: in our controllers used as case studies, the lemma is used over 50 times. So far, Anvil includes statements and proofs of 70+ such temporal reasoning lemmas, representing a broad range of temporal reasoning patterns.

4.4.2 Environment Reasoning Lemmas

Environment reasoning lemmas prove liveness and safety properties of the interaction between a controller and the environment. We have developed 60 such lemmas. These lemmas are generic to all controllers, and developers can assemble the lemmas into their proofs. We present a representative lemma derived from Anvil's Kubernetes environment model.

Example lemma on the garbage collector (GC). Developers need to reason about their controller's interaction with the built-in GC (§4.3.1). The GC's job is to delete orphan objects whose owner [21] no longer exists: e.g., a stateful set owns a set of pods, thus deleting the stateful set orphans these pods. The GC can conflict with the controller: (1) after the controller updates the owner of an orphan object, the GC deletes the object due to its stale view [86], and (2) the controller attempts to update an object that was deleted by the GC.

To prove ESR, developers need to prove that eventually the GC stops racing with the controller on the object. To help developers prove that eventually the GC stops trying to delete an object x (as x has an existing owner), Anvil provides a lemma with the precondition that any request from the controller that tries to (re)create or update x sets x's owner to an existing object, and the postcondition that eventually if x exists, it has an existing owner (Figure 12). This lemma saves developers the trouble of reasoning about a long chain of the GC execution, including that the GC eventually sends a request to delete x (if it is an orphan), the network eventually delivers the request, and the API server eventually handles the deletion. This lemma takes 200+ lines of proof code and is used in verifying all of the controllers in §6.

```
proof fn eventually_always_has_an_existing_owner(
  model: TemporalPredicate, x: ObjectKey
)
  requires model.entails(
    always(each_req_sets_an_existing_owner(x))),
    ... // some preconditions on fairness are omitted
  ensures model.entails(
    eventually(always(has_an_existing_owner(x))))
{ ... } // proof body is omitted
```

Figure 12: **The garbage collector lemma.** If each request that tries to create or update x sets x's owner to an existing object, then eventually it is always true that if x exists, it has an existing owner.

5 Proving the ESR Theorem

Proving the ESR theorem requires developers to reason about how the controller makes progress starting from any cluster state towards any desired state. We leverage the opportunity that all controllers follow the state-reconciliation principle and develop a proof strategy for ESR. The proof strategy is realized by temporal reasoning using Anvil's TLA embedding and lemmas. We present the proof strategy for ESR and temporal reasoning with Anvil in detail.

5.1 Proof Strategy for ESR

The key idea of our proof strategy is to divide the proof into two main lemmas by separation of concerns: (1) proving that the environment eventually gets stable, and (2) proving that the controller, starting from *any* state (`any_state()`) resulted from arbitrary previous executions and faults, eventually achieves the desired state in this stable environment. Here an environment is stable if (1) the controller does not conflict with the other controllers, (2) faults do not happen, and (3) the desired state description remains unchanged. The ESR theorem is finally proved by combining the two lemmas using temporal proof rules (e.g., leads-to transitivity). Figure 13 shows the high-level proof structure.

5.1.1 Environment is Eventually Stable

Proving that the environment is eventually stable is straightforward and is largely automated by Anvil's lemmas. For example, developers can directly invoke Anvil's lemma which proves that faults eventually stop happening based on Anvil's assumption of faults (§4.3.3). However, proving that the controller eventually stops conflicting with the other controllers still requires certain controller-specific reasoning. Take the garbage collector (GC) as an example, developers can use

```
proof fn ESR_proof()
  ensures model.entails(forall(|d: DV|
    always(desire(d)).leads_to(always(match(d)))
  )) /* the ESR theorem */ {
  // (1) prove ∀d.model ⊨ □desire(d) ⇝ stable_model(d)
  env_is_eventually_stable();
  // (2) prove ∀d.stable_model(d) ⊨ any_state() ⇝ □match(d)
  liveness_in_stable_env();
  // (3) prove model ⊨ ∀d.□desire(d) ⇝ □match(d)
  ...
  leads_to_transitive(...);
}

proof fn env_is_eventually_stable() // lemma 1
  ensures forall |d| model.entails(
    always(desire(d)).leads_to(stable_model(d))) {...}

proof fn liveness_in_stable_env() // lemma 2
  ensures forall |d| stable_model(d).entails(
    any_state().leads_to(always(match(d)))) {...}
```

Figure 13: **High-level structure of the ESR proof.** `model` describes the original environment in §4.3. `stable_model`(d) describes the stable environment: faults and conflicts stop, and the desired state d is stable.

Anvil's lemma on the GC (Figure 12) to prove that the GC eventually stops racing with the controller on any object, after they prove that the controller correctly sets the owner of the target objects (required by the GC lemma).

A notable corner case emerges due to asynchrony: even if the desired state description remains unchanged, the controller could still be affected by an older version of the desired state. Consider an execution where the controller crashes right after sending a request to match d_1, then the desired state description is updated to d_2 and remains unchanged from then, but the old request for d_1 is still pending in the network. After the restarted controller sends a new request to match d_2, the two requests will conflict with each other—the two requests try to make the cluster state match two different versions of the desired state. To address this problem, we prove that after the desired state description eventually stabilizes, any controller request for any previous version of the desired state will eventually leave the network.

5.1.2 Liveness in a Stable Environment

Within the stable environment, developers focus on proving that the controller reaches the desired state through each reconciliation step, without considering faults or conflicts.

The main challenge is to prove liveness starting from any possible state. The state here includes both the shared cluster state and the controller's internal state: the cluster state can result from any possible interleaving between the controller's previous execution and arbitrary faults, and the controller internally can be running any reconciliation step.

It is tedious to reason about different executions starting from every internal state. For the ZooKeeper controller in Figure 4, it would require reasoning about controller executions starting from `CheckService`, `ReconcileService` and all other branches in `step()`, respectively. To reduce proof burden, we organize the proof in three stages (Figure 14). First, we

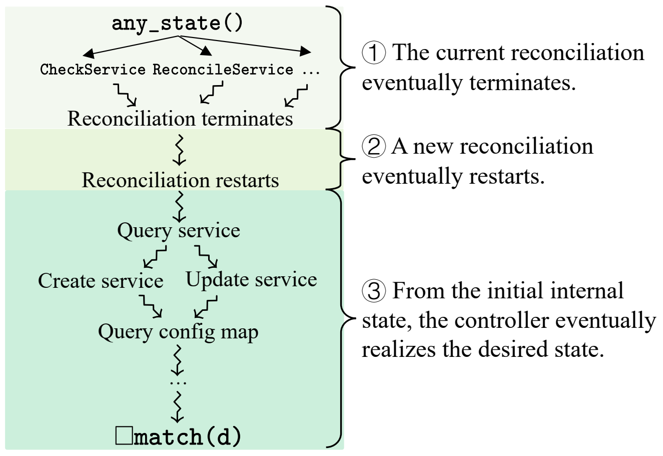

any_state()
CheckService ReconcileService ...
Reconciliation terminates
Reconciliation restarts
Query service
Create service
Update service
Query config map
...
□match(d)
① The current reconciliation eventually terminates.
② A new reconciliation eventually restarts.
③ From the initial internal state, the controller eventually realizes the desired state.

Figure 14: **Proving liveness in a stable environment.**

prove a termination property: the controller's current reconciliation (the current invocation of `reconcile()` in Figure 3) eventually terminates regardless of its current internal state. This is done by reasoning about internal states backward, e.g., `CheckService` leads to termination if all its successor states lead to termination. Second, we prove that a new reconciliation eventually starts after the previous one terminates. This holds as Anvil requeues the next invocation of `reconcile()` when the current terminates (Figure 3). Lastly, we only need to reason about the controller execution starting from its initial internal state in the new reconciliation (e.g., `CheckService` in Figure 4) to prove that the controller eventually creates and updates all the state objects to match the desired state.

To reason about the controller execution starting from its initial internal state, we need to reason about how the controller manages each state object. We observe that controllers often employ similar workflow for managing different objects, which can be leveraged to develop general lemmas to further reduce proof burden. For example, the ZooKeeper controller in Figure 4 manages its service, config map and stateful set with a similar pattern: (1) querying the object and (2) creating or updating the object depending on the query result. We develop a lemma parameterized by state objects which proves that, from the step that the controller queries the object, eventually the object always exists and matches the desired state. The lemma internally reasons about how the controller creates or updates the object to match the desired state.

5.2 Temporal Reasoning with Anvil

The proof strategy for ESR is realized by temporal reasoning. With Anvil, developers perform temporal reasoning by focusing on reasoning about state and action predicates using Anvil's TLA embedding and lemmas. We use the example in Figure 15 to demonstrate temporal reasoning with Anvil.

Developers perform temporal reasoning to prove that *all* possible executions allowed by a model satisfy a property *Prop* (`model` $\models$ *Prop*). A model is defined as the initial state

```
1  // model ≜ init∧□next∧fairness(...)
2  let model = lift(init).and(always(lift(next))
3    .and(fairness(...)));
4
5  // (1) prove model ⊨ P ⇝ Q
6  // if P holds, P or Q will hold in the next state
7  assert forall |s, s'| P(s) && next(s, s')
8  implies P(s') || Q(s') by { ... }
9  // if P holds, running A makes Q hold in the next state
10 assert forall |s, s'| P(s) && next(s, s') && A(s, s')
11 implies Q(s') by { ... }
12 // if P holds, A is enabled (A can possibly occur)
13 assert forall |s| P(s) implies enabled(A)(s) by { ... }
14 wf1(model, next, A, P, Q);
15
16 // (2) prove model ⊨ Q ⇝ R
17 ...
18 wf1(model, next, A, Q, R);
19
20 // (3) prove model ⊨ P ⇝ R
21 leads_to_transitive(model, lift(P), lift(Q), lift(R));
22
23 // (4) prove model ⊨ P ⇝ □R
24 assert forall |s, s'| R(s) && next(s, s')
25 implies R(s') by { ... }
26 leads_to_stable(model, lift(next), lift(P), lift(R));
27
28 // (5) prove model ⊨ □Inv
29 assert forall |s| init(s) implies Inv(s) by { ... }
30 assert forall |s, s'| Inv(s) && next(s, s')
31 implies Inv(s') by { ... }
32 invariant_by_induction(model, init, next, Inv);
```

Figure 15: **Temporal reasoning with Anvil.** Developers focus on reasoning about states and actions and applying TLA proof rules.

(`init`), all possible next-state actions (`next`), and fairness assumptions (line 2-3). Fairness assumptions are only used for proving liveness properties such as ESR.

Proving ESR often involves proving that if condition P holds then eventually Q holds (i.e., $P \leadsto Q$). For example, if the controller sends a request, then eventually the request is received and handled by the API server. Proving $P \leadsto Q$ is typically done by applying the WF1 rule [58]. WF1 states that "Action A makes P lead to Q" with four requirements (1) running any action in a state satisfying P makes either P or Q hold in the next state, (2) running A in a state satisfying P makes Q hold in the next state, (3) P implies that A is enabled (i.e., A can possibly occur) and (4) A has the weak fairness assumption. To apply Anvil's `wf1` lemma (line 14), developers focus on proving (1)-(3) by reasoning about P, Q, A and all other actions allowed by the model (line 7-13), and (4) is automatically proved by the definition of the model.

Proving ESR requires reasoning about a sequence of actions. For example, the controller sends a request, the API server handles the request, and the controller receives the response and continues to send the next request. To prove that the controller makes progress through multiple actions, developers apply the `leads_to_transitive` lemma (line 21) to combine multiple leads-to properties into one ($P \leadsto R$).

To reason about stability (if $P \leadsto R$, then $P \leadsto \Box R$), developers need to demonstrate that R is preserved by all possible actions (if R holds, then it will hold in the next state) and apply the `leads_to_stable` lemma (line 24-26).

Proving ESR (or other properties) often requires invariant reasoning by induction (line 29-31). For example, to prove that a state object x always exists, developers need to prove an invariant that the controller *never* deletes x. Such invariants are often required when applying `wf1` and `leads_to_stable`.

6 Case Studies

We use Anvil to build three verified Kubernetes controllers for managing different applications and services (ZooKeeper, RabbitMQ, and FluentBit). For each controller, we use a mature, widely used controller as a reference (either the official Kubernetes controller of the applications or from companies that offer related products). We verify ESR for all three controllers, and a safety property of the RabbitMQ controller.

Feature parity. We aim to implement verified controllers that are feature rich with production quality. For the ZooKeeper and RabbitMQ controllers, we implement key features offered by the reference controllers [20, 23] including scaling, version upgrading, resource allocation, pod placement, and configurations, as well as network and storage management. For the FluentBit controller, we implement *all* the features offered by the reference controller [19]. We also implement important features missing in the reference controllers. For the ZooKeeper controller, we implement a feature that the controller automatically restarts each ZooKeeper server to load the new configuration once the configuration changes. For the FluentBit controller, we implement a feature that the controller allows users to customize how a load balancer discovers FluentBit daemons. All the verified controllers can readily be deployed in real-world Kubernetes platforms and manage their respective applications.

Experience. Anvil's Controller API (Figure 2) is expressive to implement all the features of the controllers. For verification, we spent around two person-months on verifying ESR for the ZooKeeper controller, during which we developed the proof strategy (§5). We took much less time (around two person-weeks) to verify the other two controllers using the same proof strategy and similar invariants. We find Anvil's ability to formally verify a controller's implementation invaluable. We discovered deep bugs via verification. Some of them also exist in the reference controllers but were not detected by testing [44, 85].

6.1 ZooKeeper Controller

We implement and verify a full-fledged ZooKeeper controller, using the controller [23] from Pravega [22] as the reference. Figure 4 is a simplified version of our ZooKeeper controller. We discuss two challenges of verifying the controller.

Supporting non-Kubernetes APIs. We extended Anvil to support non-Kubernetes APIs to implement features like scaling. To scale a ZooKeeper cluster, the controller needs to change ZooKeeper membership by invoking ZooKeeper APIs. We implement procedures to invoke ZooKeeper APIs as callbacks invoked by `reconcile()` (Figure 3); Anvil decides whether to invoke Kubernetes APIs or ZooKeeper APIs based on the request object returned by the controller `step()`.

Invoking ZooKeeper APIs needs new specifications beyond what Anvil supplies. Hence, we write a trusted model (an abstract state machine) of the ZooKeeper APIs used by our controller and register it with the extensible compound state machine. To prove liveness, we assume weak fairness on the ZooKeeper API model: if the controller sends a request to a deployed ZooKeeper cluster, it eventually receives a response.

Reasoning about dependencies between state objects. To prove ESR, we need to reason about *dependencies* between state objects—the desired state of one object depends on the current state of another object. For example, to support reconfiguration, our controller attaches the version number of the config map to the stateful set as an annotation [7]. To ensure the ZooKeeper servers managed by the stateful set use the updated configuration, the desired state of the stateful set should contain the current version number of the config map as an annotation. To verify the correctness of reconfiguration, in ESR, `match` asserts that each state object matches the desired state description (as in Figure 10), and the annotation in the stateful set matches the current version of the config map. We prove that the config map's version eventually becomes stable and thus the annotation eventually matches the version.

Bugs precluded. We found and fixed two liveness bugs when verifying our ZooKeeper controller. The first bug occurs when the controller crashes between the steps of scaling ZooKeeper and cannot continue reconciliation after restart, similar to Figure 1. This led us to find a similar bug in the reference controller we reported in [26]. Recent work [85] applied extensive fault-injection testing on this controller but failed to find this bug, because the bug only manifests in specific timing under specific workloads (not covered by tests).

The second bug was caused by the controller trying to update immutable fields in a stateful set. Kubernetes always rejects the update, so the controller never finishes its reconciliation. Our environment model captures how Kubernetes validates each request (§4.3), which helped us find this bug.

6.2 RabbitMQ Controller

We implement and verify a full-fledged controller for RabbitMQ, a widely used message broker [24]. We use the official RabbitMQ controller as the reference [20].

Verifying safety. Besides ESR, we verify a safety property for our controller. The official RabbitMQ controller disallows scaling down a RabbitMQ cluster by reducing the stateful set's `replicas` due to data loss concerns [25]. The recommended practice is to export the data, redeploy RabbitMQ with fewer replicas, and import the data back. So, our controller prevents reducing `replicas` count. We prove a safety property stating that the replica count *never* decreases using Anvil. The safety proof is done by standard inductive proof. For example, we first prove invariants like "no request in the network reduces `replicas`," and conclude the `replicas` in the data store never decreases using the invariants.

Bugs precluded. We found a safety bug and a liveness bug via verification. The safety bug was caused by a concurrency issue involving the RabbitMQ controller and the Kubernetes garbage collector (GC). Initially, we restricted that `replicas` never decreases in desired state descriptions using Kubernetes' validation rule [5]. However, safety can still be violated, because the GC may not immediately remove orphan stateful sets. If the stateful set updated by the controller was created by an old (already deleted) desired state description that set a larger `replicas` (r_1) than the current one (r_2), the controller would in fact decrease the stateful set's `replicas` ($r_1 \rightarrow r_2$). We fixed the bug by enforcing the controller to wait for the GC to delete orphan stateful sets.

The liveness bug was caused by a naming rule we inherited from the reference controller. The bug causes the controller to assign the same name for service objects from *different* RabbitMQ clusters. In this case, the desired state descriptions of two RabbitMQ clusters drive the controller to change each other's service object back and forth, thus neither can reach desired states stably. We caught this bug because the oscillation behavior prevented us from proving the cluster state eventually *always* matches the desired state description in the presence of another conflicting description. We fixed the bug by changing the naming schema. The same bug also exists in the reference controller.

6.3 FluentBit Controller

We implement and verify a controller for FluentBit, a popular logging and metrics service [9]. FluentBit is deployed as a group of daemons collecting and processing data on different nodes in a cluster. We use the official FluentBit controller as the reference [19] and implement *all* its features.

Incremental verification. To evaluate the efforts of maintaining an evolving controller, we first implemented and verified a basic version of the controller that deploys FluentBit daemons, and then added new features incrementally, including version upgrading, daemon placement, reconfiguration. We repaired the proof every time when a new feature was added. We find the efforts of evolving a verified controller manageable (§7.1).

7 Evaluation

We evaluate Anvil along the dimensions of verification effort (§7.1), controller correctness (§7.2) and performance (§7.3). Our evaluation shows that it is pragmatic to implement, verify and evolve practical Kubernetes controllers with Anvil.

7.1 Verification Effort

Table 1 shows the details of each verified controller we built using Anvil. Verifying each controller takes under 3 minutes in real time on a 6-core 16 GB laptop with 11 parallel threads.

	Trusted	**Exec**	**Proof**	**Time to Verify**
	(lines of source code)			(seconds)
ZooKeeper controller §6.1				
Liveness (ESR)	94	–	7245	511
Conformance	5	–	172	9
Controller model	–	–	935	–
Controller implementation	–	1134	–	–
Trusted wrapper	514	–	–	–
Trusted ZooKeeper API	318	–	–	–
Trusted entry point	19	–	–	–
Total	950	1134	8352	520 (154)
RabbitMQ controller §6.2				
Liveness (ESR)	144	–	5211	278
Safety	22	–	358	45
Conformance	5	–	290	18
Controller model	–	–	1369	–
Controller implementation	–	1598	–	–
Trusted wrapper	358	–	–	–
Trusted entry point	19	–	–	–
Total	548	1598	7228	341 (151)
FluentBit controller §6.3				
Liveness (ESR)	115	–	7079	337
Conformance	10	–	201	10
Controller model	–	–	1115	–
Controller implementation	–	1208	–	–
Trusted wrapper	679	–	–	–
Trusted entry point	24	–	–	–
Total	828	1208	8395	347 (96)
Total (all)	2326	3940	23975	1208 (401)

Table 1: **Code sizes and verification time of the controllers verified using Anvil.** Trusted includes the (verified) theorems, trusted assumptions and unverified implementation. Time in brackets is obtained by running the verifier in parallel (11 threads on 6 cores).

87% of proof functions verify in under ten CPU seconds, and the slowest of them takes 120 CPU seconds.

Implementing and verifying each controller takes around 2.5 person-months. The proof-to-code ratio ranges from 4.5 to 7.4 across three controllers. We attribute the relatively low ratio to Anvil's reusable lemmas (§4.4) and our proof strategy (§5). For example, the ESR proof of the RabbitMQ controller uses the same set of leads-to reasoning lemmas to prove nine different state objects eventually match the desired state.

The ESR proof mainly consists of proving invariants and applying temporal proof rules. Proving invariants takes about 40% of the proof, which can potentially benefit from research on inductive invariant inference [42, 48, 68, 69, 78, 79, 91, 93]. All our temporal logic reasoning is done by applying Anvil's temporal logic lemmas without unfolding the definition of executions and temporal logic operators.

The verified controllers have a large portion of unverified (trusted) components: 67% of the trusted code is for defining wrapper types of Kubernetes custom objects (used for describing desired states) to integrate kube-rs, and their views to enable verification (§4.1). The ZooKeeper controller also relies on the trusted ZooKeeper API: 180 lines for specifying the ZooKeeper API and 138 lines for implementing the callbacks for Anvil to invoke the ZooKeeper API during runtime.

Controller	Functional testing		Crash testing	
	# Tests	# Bugs	# Tests	# Bugs
ZooKeeper	239	1	212	0
RabbitMQ	197	0	158	0
FluentBit	557	0	484	0

Table 2: **Testing results of the three verified controllers.** The tests cover all the features of the controller under test.

Controller	Verified (Anvil)		Reference (unverified)	
	Mean (ms)	Max (ms)	Mean (ms)	Max (ms)
ZooKeeper	439	696	212	413
RabbitMQ	439	725	690	1531
FluentBit	195	303	221	464

Table 3: **Comparison of `reconcile()` execution time (in milliseconds) between the verified controllers and their references.**

Evolving controllers with Anvil. We measure the efforts to evolve the FluentBit controller with Anvil by incrementally adding features and updating its proof. We first implemented and verified a basic FluentBit controller for deploying FluentBit daemons, then added 28 new features including version upgrading, daemon placement, and various configurations. On average, implementing a feature took less than a day and 47 lines of changes, including 19 lines in the proof. Among them, implementing `metrics_port` required the most changes (403 lines in total and 211 in the proof); it added a new service that routes traffic to the metrics port of each daemon, and we proved the service eventually matches the desired state.

Effort to build Anvil. As a reference, the Anvil framework consists of 5353 lines of reusable lemmas and 7817 lines of trusted code, including the TLA embedding (85 lines), the environment model (1846 lines) and the integration with Kubernetes (5886 lines); 89% of the integration is for defining wrapper types and views of Kubernetes built-in objects (§4.1). All the lemmas are verified in under one minute.

7.2 Controller Correctness

We run extensive end-to-end functional tests on the verified controllers using Acto [44]. Acto generates different desired state descriptions to exercise controller reconciliation under various scenarios. We also run extensive crash tests to check if the verified controllers can recover from random crashes during their reconciliation. The crashes are injected using an implementation of Sieve [85] for Rust controllers.

Table 2 shows the testing results. The crash tests did not find any bug—the verified controllers correctly recovered from all the injected crashes and successfully reconciled the cluster to the desired state. The functional tests found a bug in the ZooKeeper controller (no bug found in other controllers).

The bug is caused by an incomplete specification of a trusted ZooKeeper API that did not cover ZooKeeper misconfigurations. If a misconfiguration results in partial failures (ZooKeeper is still running but cannot serve write requests [67]), the controller fails to update the membership and thus blocks the subsequent reconciliation steps. We fixed this bug by adding configuration validation in the implementation, enhancing the specification, and updating the proofs.

7.3 Controller Performance

The verified controllers have comparable performance to the reference controllers. We use Acto [44] to generate many different desired state descriptions, triggering a sequence of reconciliations. For each desired state, we measure (1) execution times for the target controllers' `reconcile()` methods (Figure 1), and (2) the time it takes for the system to be fully reconciled (e.g., after the controller issues a rolling update). The experiments are run on CloudLab Clemson c6420 machines with dual Intel Xeon Gold 6142 processors, 384GB DRAM, and a 6Gb/s HDD running Ubuntu 20.04 LTS.

Table 3 shows that the verified and reference controllers have comparable execution times. The verified ZooKeeper controller's execution time is about twice that of the reference which implements optimizations to conditionally skip state updates. None of the controllers are latency critical. On average, `reconcile()`'s execution time takes less than 1% of the overall system reconciliation time, most of which is out of the control of the controller (e.g., container restart time).

We also evaluate if the verified controllers introduce more load on the data store which is often the bottleneck for Kubernetes scalability [28, 87]. We measure the disk I/O of etcd and the verified controllers do not cause noticeably more loads—the verified FluentBit controller causes only 0.44% load increase compared to the reference; the other two verified controllers do not cause load increase.

8 Related Work

Anvil is the first effort for building formally verified cluster management controllers. We discuss related work in controller correctness, systems verification and liveness verification.

Controller correctness. Liu et al. [65] use model checking to find if controllers in a specific deployment have conflicting interactions that violate user-supplied policies at the model level (not executables). In contrast, Anvil verifies controller *implementations* against ESR, a *general* controller-correctness specification. Automated testing techniques [44, 85] have found bugs in controller implementations. Anvil precludes such bugs by verifying that the controller implementation satisfies ESR for all executions. It has also revealed bugs that were missed by these automated testing techniques (§6).

Systems verification. Despite the rich literature, most systems verification efforts so far focus on safety rather than liveness [31–34, 36, 37, 49, 50, 56, 63, 66, 75, 82, 83, 88, 90, 94]. A notable exception is IronFleet [51], which also verifies liveness of system implementations.

Anvil differs from IronFleet in the objective and proof technique. Regarding objective, IronFleet verifies a Paxos-based replicated state machine and a sharded key-value store, with system-specific specification (e.g., “if the network is fair then the reliable-transmission component eventually delivers each message”). Differently, Anvil formalizes ESR as a general specification that captures the essence of state reconciliation and verifies multiple controllers against ESR. Anvil shares IronFleet’s methodology of using TLA embedding on first-order logic. Different from many IronFleet’s liveness proof statements that interact directly with instantiated executions by indexing (Figure 16), Anvil abstracts away executions to let developers model components, at the level of state and action, and complete liveness proofs exclusively with temporal operators (Figure 15).

```
lemma Lemma_PacketSentEventuallyReceivedAndNotDiscarded
  (b:Behavior<LSHT_State>, send_step:int, ...)
  returns (received_step:int, ...)
  requires 0 <= send_step;
  requires SendSingleValid(b[send_step], ...);
  requires ... // other preconditions are omitted
  ensures send_step <= received_step;
  ensures b[received_step].hosts[dst_idx].host.
    receivedPacket == Some(Packet(msg, ...));
{ ... } // proof body is omitted
```

Figure 16: **A representative liveness lemma example from IronFleet (written in Dafny) [10].** The lemma counts steps in one instantiated execution (`Behavior`) to prove that if the packet is sent at `b[send_step]`, it will be received at `b[received_step]`. This lemma, if written in Anvil, will have a postcondition in the form of `model.entails(sent.leads_to(received))` without taking or returning any execution instances or indices.

Liveness verification. Ivy [72, 78] incorporates a technique for proving liveness of distributed protocols using first-order logic [76, 77]. Compared to Anvil, Ivy obtains a higher degree of proof automation at the expense of a more restricted modeling logic; we are exploring the potential to leverage some of Ivy’s techniques in Anvil. LVR [92] proves liveness of distributed protocols by automatically synthesizing ranking functions with limited manual guidance. LVR is complementary to Anvil and might be able to synthesize ESR proofs for controller implementations. The Alloy analyzer has recently been extended to support linear temporal logic [3, 29, 70], which enables modeling liveness properties of protocols and system abstractions; but only finite instances can be checked and the analyzed abstractions are not formally linked to executable code. More broadly, the rich literature on liveness verification includes program termination [38] and liveness of concurrent programs [27, 41, 43]. These techniques target other systems and their liveness specifications, whereas Anvil’s contribution specifically targets controller correctness and connects liveness proofs to an executable implementation.

9 Discussion and Future Work

The correctness of controllers verified by Anvil is not absolute. Anvil relies on trusted components, including the model of the environment, the shim layer, trusted external APIs, and the verifier, compiler, and OS. We indeed found a bug caused by an incomplete trusted assumption (§7.2). We believe that the bug does not diminish the value of Anvil. Anvil formally verifies reconciliation – the core of a controller – and reduces the code one needs to look for bugs in to the trusted assumptions.

Note that ESR does not preclude all possible controller bugs. For example, ESR may not rule out all potential safety violations. Unlike ESR as a *general* correctness specification, safety properties are often controller-specific; e.g., the safety property we verified in §6.2 that the replicas number never decreases is specific to the RabbitMQ controller.

We choose to focus on verifying ESR because ESR is a general, reusable property that precludes a broad range of bugs, and it is straightforward for developers to specify ESR. Some bugs precluded by ESR may be precluded by some safety properties as well, but these safety properties may be more difficult for developers to specify. For example, the bug in Figure 1 could be precluded by a safety property saying “irrecoverable intermediate states never happen.” However, specifying such safety properties requires knowledge of the nature of the bugs (e.g., what kind of intermediate states the controller cannot recover from?) [55]. In contrast, specifying ESR only requires knowledge of desired states.

We expect verified controllers to be deployed on real-world Kubernetes platforms, running alongside unverified controllers. If the unverified controllers are custom controllers not modeled in Anvil (§4.3.1), Anvil cannot reason about their interactions with verified controllers, and hence cannot rule out bugs caused by conflicting interactions.

In future work, we aim to gradually replace existing (unverified) controllers with verified controllers using Anvil, including both custom and built-in ones. We plan to extend Anvil to admit multiple verified controllers and verify the interactions among them in a modular way. We also plan to ensure the quality of the trusted model of the environment, the shim layer, and external APIs using lightweight formal methods.

10 Concluding Remarks

This paper presents Anvil, a framework for developing and verifying cluster-management controllers, and ESR, a general specification for controller correctness. We have implemented and verified three Kubernetes controllers using Anvil. Our work shows that it is not only feasible but also pragmatic to implement, verify, and maintain practical Kubernetes controllers. We hope that Anvil and ESR lead to a practical path towards provably correct cloud infrastructures.

Acknowledgement

We thank the anonymous reviewers for their insightful comments on the paper. We thank Jay Lorch, Owolabi Legunsen, Ramnatthan Alagappan, and Yongle Zhang for their valuable feedback and discussions that helped shape this work. This work was funded in part by NSF CNS-2145295 and a VMware Research Gift.

References

[1] etcd. https://etcd.io/.

[2] Flink controller state machine. https://github.com/lyft/flinkk8soperator/blob/master/docs/state_machine.md, 2020.

[3] Alloy 6. https://alloytools.org/alloy6.html, 2021.

[4] Controllers and Reconciliation. https://cluster-api.sigs.k8s.io/developer/providers/implementers-guide/controllers_and_reconciliation.html, 2021.

[5] CustomResourceDefinition Validation Rules. https://kubernetes.io/blog/2022/09/23/crd-validation-rules-beta/, 2022.

[6] Spark controller state machine. https://github.com/GoogleCloudPlatform/spark-on-k8s-operator/blob/master/pkg/controller/sparkapplication/controller.go#L485-L520, 2022.

[7] Annotations. https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/, 2023.

[8] Controllers. https://kubernetes.io/docs/concepts/architecture/controller/, 2023.

[9] FluentBit. https://fluentbit.io/, 2023.

[10] Ironfleet liveness lemma. https://github.com/microsoft/Ironclad/blob/2fe4dcdc323b92e93f759cc3e373521366b7f691/ironfleet/src/Dafny/Distributed/Protocol/LiveSHT/LivenessProof/LivenessProof.i.dfy#L31, 2023.

[11] kube-rs/kube: Rust Kubernetes client and controller runtime. https://github.com/kube-rs/kube, 2023.

[12] Kubernetes. https://kubernetes.io/, 2023.

[13] Kubernetes API Concepts: Updates to existing resources. https://kubernetes.io/docs/reference/using-api/api-concepts/#patch-and-apply, 2023.

[14] Kubernetes ConfigMaps. https://kubernetes.io/docs/concepts/configuration/configmap/, 2023.

[15] Kubernetes DaemonSet. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/, 2023.

[16] Kubernetes Garbage Collection. https://kubernetes.io/docs/concepts/architecture/garbage-collection/, 2023.

[17] Kubernetes Service. https://kubernetes.io/docs/concepts/services-networking/service/, 2023.

[18] Kubernetes StatefulSet. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, 2023.

[19] Official FluentBit controller. https://github.com/fluent/fluent-operator, 2023.

[20] Official RabbitMQ controller. https://github.com/rabbitmq/cluster-operator, 2023.

[21] Owners and Dependents. https://kubernetes.io/docs/concepts/overview/working-with-objects/owners-dependents/, 2023.

[22] Pravega. https://cncf.pravega.io/, 2023.

[23] Pravega ZooKeeper controller. https://github.com/pravega/zookeeper-operator, 2023.

[24] RabbitMQ. https://www.rabbitmq.com/, 2023.

[25] RabbitMQ: Scale down. https://github.com/rabbitmq/cluster-operator/issues/223, 2023.

[26] Stateful set never gets updated because zk node is missing. https://github.com/pravega/zookeeper-operator/issues/569, 2023.

[27] BAUMANN, P., MAJUMDAR, R., THINNIYAM, R. S., AND ZETZSCHE, G. Context-Bounded Verification of Liveness Properties for Multithreaded Shared-Memory Programs. In *Proceedings of the 48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'21)* (Jan. 2021).

[28] BERNER, C. Scaling kubernetes to 2,500 nodes. https://openai.com/research/scaling-kubernetes-to-2500-nodes, Jan. 2023.

[29] BRUNEL, J., CHEMOUIL, D., CUNHA, A., AND MACEDO, N. The Electrum Analyzer: Model Checking Relational First-Order Temporal Specifications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)* (Sept. 2018).

[30] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, Omega, and Kubernetes. *Communications of the ACM 59*, 5 (May 2016), 50–57.

[31] CHAJED, T., TASSAROTTI, J., KAASHOEK, M. F., AND ZELDOVICH, N. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[32] CHAJED, T., TASSAROTTI, J., THENG, M., JUNG, R., KAASHOEK, M. F., AND ZELDOVICH, N. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)* (July 2021).

[33] CHAJED, T., TASSAROTTI, J., THENG, M., KAASHOEK, M. F., AND ZELDOVICH, N. Verifying the DaisyNFS concurrent and crash-safe file system with sequential reasoning. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[34] CHANG, Y.-S., JUNG, R., SHARMA, U., TASSAROTTI, J., KAASHOEK, M. F., AND ZELDOVICH, N. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)* (July 2023).

[35] CHEKRYGIN, I. Keep the Space Shuttle Flying: Writing Robust Operators. https://youtu.be/uf97lOApOv8?t=1457, May 2019.

[36] CHEN, H., CHAJED, T., KONRADI, A., WANG, S., ILERI, A., CHLIPALA, A., KAASHOEK, M. F., AND ZELDOVICH, N. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[37] CHEN, H., ZIEGLER, D., CHAJED, T., CHLIPALA, A., KAASHOEK, M. F., AND ZELDOVICH, N. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[38] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Proving Program Termination. *Communications of the ACM 54*, 5 (May 2011), 88–98.

[39] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)* (Mar. 2008).

[40] DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the Presence of Partial Synchrony. *Journal of the ACM 35*, 2 (Apr. 1988), 288–323.

[41] FARZAN, A., KINCAID, Z., AND PODELSKI, A. Proving Liveness of Parameterized Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'16)* (July 2016).

[42] GOEL, A., AND SAKALLAH, K. On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *Proceedings of the 13th NASA Formal Methods Symposium (NFM'21)* (May 2021).

[43] GOTSMAN, A., COOK, B., PARKINSON, M., AND VAFEIADIS, V. Proving That Non-Blocking Algorithms Don't Block. In *Proceedings of the 36th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'09)* (Jan. 2009).

[44] GU, J. T., SUN, X., ZHANG, W., JIANG, Y., WANG, C., VAZIRI, M., LEGUNSEN, O., AND XU, T. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP'23)* (Oct. 2023).

[45] GUILLOUX, S. Writing a Kubernetes Operator: the Hard Parts. https://youtu.be/wMqzAOp15wo?t=411, Nov. 2019.

[46] HAASE, S. How an Operator Becomes the Hero of the Edge. In *OperatorCon* (May 2019).

[47] HALL, C. 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, Mar. 2019.

[48] HANCE, T., HEULE, M., MARTINS, R., AND PARNO, B. Finding Invariants of Distributed Systems: It's a Small (Enough) World After All. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)* (Apr. 2021).

[49] HANCE, T., LATTUADA, A., HAWBLITZEL, C., HOWELL, J., JOHNSON, R., AND PARNO, B. Storage Systems are Distributed Systems (So Verify Them That Way!). In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[50] HANCE, T., ZHOU, Y., LATTUADA, A., ACHERMANN, R., CONWAY, A., STUTSMAN, R., ZELLWEGER, G., HAWBLITZEL, C., HOWELL, J., AND PARNO, B. Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)* (July 2023).

[51] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[52] HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12* (1969).

[53] HOCKIN, T. Kubernetes: Edge vs. Level Triggered Logic. https://speakerdeck.com/thockin/edge-vs-level-triggered-logic, June 2017.

[54] HOWARD, J. Building Better Controllers. https://www.youtube.com/watch?v=GKPBQDJ2Hjk&t=160s, Nov. 2023.

[55] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)* (Apr. 2007).

[56] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)* (Oct. 2009).

[57] LAGRESLE, M. Moving to Kubernetes: the Bad and the Ugly. https://youtu.be/MoIdU0J0f0E?t=263, June 2019.

[58] LAMPORT, L. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 872–923.

[59] LAMPORT, L. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[60] LANDER, R. Kubernetes Operators: Should You Use Them? https://tanzu.vmware.com/developer/blog/kubernetes-operators-should-you-use-them/, July 2021.

[61] LATTUADA, A., HANCE, T., CHO, C., BRUN, M., SUBASINGHE, I., ZHOU, Y., HOWELL, J., PARNO, B., AND HAWBLITZEL, C. Verus: Verifying Rust Programs Using Linear Ghost Types. In *Proceedings of 2023 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'23)* (Apr. 2023).

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

[63] LEROY, X. Formal Verification of a Realistic Compiler. *Communications of the ACM 52*, 7 (July 2009), 107–115.

[64] Liu, B., Kheradmand, A., Caesar, M., and Godfrey, P. B. Towards Verified Self-Driving Infrastructure. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)* (Nov. 2020).

[65] Liu, B., Lim, G., Beckett, R., and Godfrey, P. B. Kivi: Verification for Cluster Management. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC'24)* (July 2024).

[66] Lorch, J. R., Chen, Y., Kapritsos, M., Parno, B., Qadeer, S., Sharma, U., Wilcox, J. R., and Zhao, X. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)* (June 2020).

[67] Lou, C., Huang, P., and Smith, S. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (Feb. 2020).

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

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

[70] Macedo, N., Brunel, J., Chemouil, D., and Cunha, A. Pardinus: A temporal relational model finder. *J. Autom. Reason. 66*, 4 (2022), 861–904.

[71] Madhu, C. Preventing Controller Sprawl From Taking Down Your Cluster. https://youtu.be/fu5GXo7jmV0?t=732, Oct. 2022.

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

[73] Melissaris, T., Nabar, K., Radut, R., Rehmtulla, S., Shi, A., Chandrashekar, S., and Papapanagiotou, I. Elastic Cloud Services: Scaling Snowflake's Control Plane. In *Proceedings of the 13th ACM Symposium on Cloud Computing (SOCC'22)* (Nov. 2022).

[74] Motwani, S., and Maheshwari, A. Deep Dive Into Writing a Kubernetes Operator: Let's Avoid Data Loss and Down Times. https://www.youtube.com/watch?v=2NjMHLACvc0&t=737s, Nov. 2023.

[75] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[76] Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., and Shoham, S. Reducing Liveness to Safety in First-Order Logic. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[77] Padon, O., Hoenicke, J., McMillan, K. L., Podelski, A., Sagiv, M., and Shoham, S. Temporal Prophecy for Proving Temporal Properties of Infinite-State Systems. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design (FMCAD'18)* (Oct. 2018).

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

[79] Padon, O., Wilcox, J. R., Koenig, J. R., McMillan, K. L., and Aiken, A. Induction Duality: Primal-Dual Search for Invariants. In *Proceedings of the 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'22)* (Jan. 2022).

[80] Pnueli, A. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (Oct. 1977).

[81] Ratis, P. Lessons Learned using the Operator Pattern to build a Kubernetes Platform. In *USENIX SREcon* (Oct. 2021).

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

[83] Sigurbjarnarson, H., Bornholt, J., Torlak, E., and Wang, X. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[84] Sosa, C., and Bhatia, P. Application management made easier with Kubernetes Operators on GCP Marketplace. https://cloud.google.com/blog/products/containers-kubernetes/application-management-made-easier-with-kubernete-operators-on-gcp-marketplace, May 2019.

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

[86] Sun, X., Suresh, L., Ganesan, A., Alagappan, R., Gasch, M., Tang, L., and Xu, T. Reasoning about modern datacenter infrastructures using partial histories. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)* (May 2021).

[87] Tang, C., Yu, K., Veeraraghavan, K., Kaldor, J., Michelson, S., Kooburat, T., Anbudurai, A., Clark, M., Gogia, K., Cheng, L., Christensen, B., Gartrell, A., Khutornenko, M., Kulkarni, S., Pawlowski, M., Pelkonen, T., Rodrigues, A., Tibrewal, R., Venkatesan, V., and Zhang, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the*

14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20) (Nov. 2020).

[88] TAUBE, M., LOSA, G., MCMILLAN, K. L., PADON, O., SAGIV, M., SHOHAM, S., WILCOX, J. R., AND WOOS, D. Modularity for Decidability of Deductive Verification with Applications to Distributed Systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)* (June 2018).

[89] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)* (Apr. 2015).

[90] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)* (June 2015).

[91] YAO, J., TAO, R., GU, R., AND NIEH, J. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[92] YAO, J., TAO, R., GU, R., AND NIEH, J. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. In *Proceedings of the 51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'24)* (Jan. 2024).

[93] YAO, J., TAO, R., GU, R., NIEH, J., JANA, S., AND RYAN, G. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)* (July 2021).

[94] ZOU, M., DING, H., DU, D., FU, M., GU, R., AND CHEN, H. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).