VERIFICATION AND TESTING OF CLOUD INFRASTRUCTURE SYSTEMS

BY

XUDONG SUN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2025

Urbana, Illinois

ABSTRACT

Cloud infrastructure systems like Kubernetes, Borg, and Twine are the foundation of the modern cloud computing world. These systems are architected as a fleet of controllers. Controllers manage large-scale cluster resources and all applications running atop them, making their reliability paramount. Bugs in controllers can affect all the upper layer applications and lead to severe consequences, such as service outages, data loss, and security issues. Ensuring the reliability of controllers is notoriously hard as controllers perform sophisticated management tasks while running within complex and dynamic environments.

This dissertation focuses on improving the reliability of cloud infrastructure systems using formal verification and testing techniques: Formal verification enables a path toward fully verified cloud infrastructure systems by incrementally replacing existing controllers with verified ones, and continuous and extensive testing improves controller reliability before there are verified replacements.

We first present state-centric reasoning, a general approach to reasoning about the behaviors of controllers. The key idea is to reason about the cluster state shared by controllers, instead of each controller's internal state. State-centric reasoning represents a controller's behavior as the cluster state's evolution—a uniform representation for diverse controllers. The uniform representation enables formal verification and efficient testing for controllers.

As the first step to formally verify controllers, we present eventually stable reconciliation (ESR), a general formal specification for controller correctness. The key idea of ESR is to capture state reconciliation, the essential functionality of controllers, using a liveness property that describes how the cluster state should evolve. We formalize ESR as a concise formula in TLA. ESR is powerful enough to preclude a broad range of controller bugs and is realistic with appropriate assumptions on the environment.

To close the gap between formal specifications (e.g., ESR) and controller implementation code, we present Anvil, the first framework that allows developers to build formally verified, practical controller implementations. Anvil emphasizes verifying both liveness and safety properties for implementation code. To achieve this goal, Anvil combines Hoare-style reasoning for imperative code and TLA-style reasoning for state machines. To reduce the manual proof burden, Anvil provides verification support, including reusable models and lemmas. With Anvil, we have built the first verified Kubernetes controllers for managing critical distributed applications. The verified controllers achieve feature parity and competitive performance compared to their unverified, mature references.

Formal verification offers strong correctness guarantees, but we still need to test existing controllers continuously and extensively before there are verified replacements. Testing can also catch bugs that originate from the interaction between verified and unverified systems. However, previous work does not offer a generally applicable, comprehensive, and efficient testing approach for diverse controllers. To address this problem, we present Sieve, the first automatic reliability testing tool for controllers. Sieve's key idea is state perturbation: Sieve perturbs the controller's view of the cluster state in ways it is expected to tolerate, and then compares the cluster state's evolution with and without perturbations to detect triggered bugs. We evaluated Sieve on ten popular open-source controllers of various kinds. Sieve found 46 new bugs in total, among which 35 have been confirmed (22 fixed) after we reported them.

This dissertation marks a first step toward building fully verified cloud infrastructure systems. We conclude by outlining future directions to advance this vision.

To my parents, for their endless love.

ACKNOWLEDGMENTS

How did I make it through these six years, transforming from a clueless undergraduate into a soon-to-be assistant professor? I was incredibly fortunate to receive unwavering support from so many people along the way. I want to express my deepest gratitude to all of them. This acknowledgment is the hardest part of my dissertation to write—my words cannot begin to capture the kindness and help I have received.

Let me begin by thanking my advisor, Tianyin Xu. When I was applying to graduate school, my application was much thinner than most others, and getting admitted to UIUC felt almost hopeless. But Tianyin believed in my potential and chose to recruit me. As one of his first students, I was fortunate to learn how an assistant professor builds a research group from the ground up. I am always amazed by how Tianyin positions each project within the broader research landscape and by how tirelessly he provides feedback to every one of his students. Tianyin has always been deeply supportive, creating a protected space where I could work freely on what I found interesting. After my first project, Tianyin encouraged me to explore new directions through an internship, which led to my second project, Sieve, at VMware Research. Later, I made a bold decision to shift my focus from testing to formal verification with Anvil, and once again, I had Tianyin's full support. Both Sieve and Anvil now form the core of this dissertation. Tianyin treated his students not just as colleagues, but also as family. When I felt lonely or lost, he and his wife, Fei, often took the time to hang out or have dinner with me. At a point during my job search when I felt hopeless and nearly wanted to give up, Tianyin and Fei invited me to their home, shared a warm meal, and helped me find optimism again. I chose to pursue an academic career not only because I love doing research, but also because I hope to become the kind of advisor Tianyin has been to me.

Having one great advisor is already a stroke of luck—I was fortunate to have two. I want to thank Lalith Suresh for mentoring me throughout my PhD. Although Lalith was never my official advisor, he always treated me as his PhD student. I began working with Lalith during my first internship at VMware Research, and from that point on, he has played a pivotal advisory role in my journey. From him, I learned both the lower-order bits and the higher-order bits of leading research projects. Although my first internship was remote due to COVID, Lalith made sure I never felt like I was working alone. He often sat with me (virtually) to debug Kubernetes failures. After the Sieve project, Lalith encouraged me to take on something riskier but potentially more rewarding. That's how I began learning formal verification and started the Anvil project. During my second internship at VMware Research, Lalith went above and beyond by assembling a dream team of experts in distributed systems and formal methods to mentor me. This dissertation would not have been possible without Lalith's mentorship.

I want to thank Darko Marinov, who has deeply influenced me as a researcher. Although we never collaborated on a research project, I consistently received valuable feedback from Darko throughout my PhD. His questions were always sharp and thought-provoking, and I'm grateful that I had the opportunity to practice addressing them from my very first year. I benefited immensely from the events Darko organized, such as the Brett Daniel Software Engineering Seminars, where I learned how to critically read and evaluate research papers, and the PhD Job Search Seminars, which provided a comprehensive view of the faculty application process. During my job search, Darko offered invaluable support—helping me improve my CV, application statements, and job talk, and coaching me on how to handle one-on-one interviews and faculty dinners. When I was feeling discouraged during job search, Darko shared his own job search experience with me and taught me how to deal with disappointment. Darko also provided valuable feedback on this dissertation. I'm incredibly grateful to have had Darko as a role model from the beginning of my PhD journey.

I also want to thank the remaining members of my thesis committee: Aishwarya Ganesan, Philip Brighten Godfrey, and Jon Howell.

I'm grateful to Aishwarya Ganesan and Ramnatthan Alagappan for both our research collaboration and their support during my job search. I began working with Aishwarya and Ram in my second year, and I have always been impressed by their sharp insights into the design of distributed systems. I'm thankful for all the technical discussions we had while working on Sieve and its follow-up projects, as well as for their thoughtful feedback on my job talk. While preparing my own talk, I watched both Aishwarya's and Ram's recorded job talks multiple times to learn how to give a top-notch job talk.

I want to thank Philip Brighten Godfrey for his insightful feedback on my dissertation work. Brighten's advice on how to articulate the synergy among the different components of my dissertation and highlight the broader applications of my research improved my job talk and dissertation. I'm also grateful to Brighten for organizing many retreat events for the SysNet group at UIUC. I thoroughly enjoyed the panel discussions, research talks, and trivia sessions—they were both intellectually stimulating and a lot of fun.

I want to thank Jon Howell for introducing me to the world of formal verification. I've always been amazed by Jon's clarity of thought, rigorous approach to proof engineering, deep understanding of both the high-level vision and the low-level details, and his ever-present sense of humor. Formal verification has a notoriously steep learning curve, and I navigated

much of that journey through the lecture materials Jon and Manos produced. The Anvil project involved writing TLA-style liveness proofs—a challenging skill—and Jon guided me through the process, once filling an entire whiteboard with annotations and formulas to share insights from the IronFleet project. He also drafted the initial TLA embedding that served as the foundation for all the liveness proofs in Anvil. Beyond the project itself, Jon provided detailed and thoughtful feedback on how to improve this dissertation.

I have received mentorship from many individuals beyond my PhD committee, and I want to sincerely thank them all, including Suman Nath, Tej Chajed, Oded Padon, Andrea Lattuada, Adriana Szekeres, Owolabi Legunsen, and Michael Gasch. Suman mentored me during my internship at Microsoft Research, where we worked on the Rainmaker project. I thoroughly enjoyed our discussions on failure patterns in emerging cloud computing paradigms. Suman was also incredibly supportive during my job search; he wrote me recommendation letters and actively looked out for opportunities for me. Tej's knack for identifying the technical essence of a verification challenge and proposing sound proof principles was always impressive. Oded's ability to formalize complex ideas in logic and explain them with intuition constantly inspired me. I'm deeply grateful that both Tej and Oded offered me gap-year postdoc positions during this challenging time. Tej also hosted my visit to the University of Wisconsin-Madison when I gave a talk at the MadSystems seminar, and his feedback on my job talk was immensely helpful. Andrea provided tremendous technical support for using Verus and was amazingly responsive to every question I had. Adriana guided me through the steep learning curve of formal verification during my internship at VMware Research. Owolabi mentored me on my first project, Ctest, and taught me how to manage junior PhD students and how to articulate the principles behind practical testing. He also gave me valuable feedback during a mock interview in my job search process. Michael's deep understanding of the Kubernetes API helped me implement the fault injection mechanism in Sieve.

I want to thank everyone who is or was a member of xlab, including Wenyu Wang, Zirui (Neil) Zhao, Qingrong Chen, Jinghao Jia, Yinfang Chen, Jiawei (Tyler) Gu, Siyuan Chai, Shuai Wang, Wentao Zhang, Jiyuan Zhang, Zhizhen (Cathy) Cai, Cody Rivera, Jackson Clark, Yiming Su, Jongyul Kim, Chaitanya Bhandari, Ran (Elaine) Ang, Jianyan (Jack) Chen, Wenqing Luo, Ruowen Qin, Ayush Bansal, Hao Lin, Yuanliang Zhang, Runxiang (Sam) Cheng, Xuhao Luo, Lilia Tang, Henry Zhu, Kaiyuan Zhang, Andrew Yoo, Kai-Hsun (Kevin) Chen, Wenjie Ma, Zicheng Ma, Shurang Wu, and others. They make xlab a family to me.

Wenyu, Neil, and Qingrong were like big brothers to me, and they helped me quickly adjust to life in Champaign. Neil gave me thoughtful feedback on my job talk and regularly

checked in to make sure everything was going smoothly. He also generously shared his job search experience with me, which helped me approach the process with confidence. Neil also provided valuable feedback on this dissertation. Qingrong introduced me to many of his friends and often invited me to lunch and dinner. After he graduated, we reconnected in the Bay Area, where he picked me up from the airport and helped me get settled for my internship. It was a great honor to be a groomsman at Qingrong and Qianying's wedding.

My fellow labmates were a source of strength that carried me through many tough moments during my PhD. Jinghao and I shared so many hobbies, and we never got tired of talking about anime. During my job search, he gave me countless rides between Champaign and Chicago—once even making a round trip with Tyler during a blizzard just to bring me back to Champaign. That trip cost them nearly half a day. Yinfang and I shared similar educational paths, and we always had a lot to talk about. We often chatted together and helped each other through tough times. Tyler and I collaborated on almost every project, and many of my submissions would have been impossible without his help. Working with Tyler was one of the most fulfilling experiences I've had. Siyuan is the most energetic person I've ever met, and his vibrancy is infectious—it inspires everyone around him. Even though I was the most senior student in the group, Yinfang, Tyler, and Siyuan often took on responsibilities that should have been mine. They organized several group events, including hotpot nights that brought us together on New Year's Eve. Hao spent less than a year visiting xlab at UIUC, but we quickly became close friends. I cherish the time we spent talking about research and anime.

I want to thank my other labmates who worked alongside me. Sam, Jack, and Elaine worked with me on my first project, Ctest, and thanks to them, I never felt like I was fighting alone during that first year. I still remember the bittersweet moments of debugging testing failures with Jack as winter approached. Lilia helped conduct a failure study that inspired Sieve. Wenqing is one of the best hackers I know and made significant contributions to Sieve—he led the project while I was interning at Microsoft Research during the summer. Wenjie and Zicheng, as undergraduate students, made valuable contributions to Anvil's success—Wenjie notably helped significantly with the proofs. Shurang and Cathy collaborated with me on a testing project. Cody and Cathy are currently working on a verification project, keeping the work going while I was occupied with my job search.

I also want to thank my other friends in Champaign, including Yifan Zhao, Shaoxiong Yao, Ke Du, Yuchen Jiang, Shizhuo (Dylan) Zhang, Bingzhe Liu, Hao Wu, and many others. Yifan was one of my first friends in Champaign and has been my roommate since my second year. I couldn't have asked for a better friend or roommate. Yifan is always kind, generous, helpful, and thoughtful. When my parents visited, Yifan generously offered us his own room and stayed with a friend so my parents and I could be more comfortable staying together. I've also lost count of how many times I enjoyed Yifan's amazing cooking—I will miss his beef stew and onion soup forever.

I have also made friends from other places during internships, conferences, and visits, including Yi Xu, Ding Ding, Sagar Patel, Jing Liu, Chenhao Ye, Yifan Dai, Anthony Rebello, Bogdan Alexandru Stoica, and others. I appreciate the time we spent together.

I want to thank my advisor and friends from my undergraduate days at Nanjing University. Yu Huang was my undergraduate advisor. Yu wrote a recommendation letter to Tianyin that helped launch my PhD journey. I am forever grateful for his support. I was also fortunate to participate in several research projects with Yu's group. Ruize Tang was my roommate during undergrad, and I was honored to be involved in his PhD dissertation. His resilience, persistence, and hard work have amazed me ever since.

I want to thank everyone who helped me during my job search. Tianyin Xu, Darko Marinov, Tej Chajed, Saugata Ghose, Aishwarya Ganesan, Ramnatthan Alagappan, Zirui (Neil) Zhao, Owolabi Legunsen, Sasa Misailovic, Ling Ren, Elahe Soltanaghai, Deepak Vasisht, Mahesh Viswanathan, Madhusudan Parthasarathy, Gagandeep Singh, Carl Gunter, and Luyi Xing have provided valuable feedback on my practice job talk and mock interview. Ding Yuan, Yongle Zhang, Chang Lou, Lin Tan, Shan Lu, Ryan Huang, Yuke Wang also offered valuable advice during the process. Shan and Suman helped me connect to opportunities at Microsoft Research. The Mavis Future Faculty Fellows Program offered helpful lectures to prepare me and other faculty candidates in every aspect.

I would also like to thank the staff members at the Siebel School of Computing and Data Science. Jennifer Comstock was incredibly responsive to all my questions regarding my final defense and thesis submission, and Ruth Anders provided valuable assistance with room bookings and reimbursements.

Finally, I want to thank my parents for their endless and unconditional love. Pursuing a PhD is a privilege, and I am profoundly grateful for their unwavering support throughout this journey. I know my parents have always been thinking of me and often worried about me—especially during the difficult times of COVID. I was able to get through the darkest moments because I knew they would accept me no matter what, and I have always wanted to make them proud. My parents also made many sacrifices during my PhD. Unlike most families, we couldn't reunite during Chinese New Years, and they endured over 20 hours of international travel just to visit me. Each time I said goodbye to them at the airport, it felt like my heart was being torn apart. I still remember my mother crying at Beijing airport when she sent me off to the US, and my father tearing up when we parted ways at Boston airport—the first time I had ever seen him cry.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Modern cloud computing world depends crucially on cloud infrastructure systems, such as Kubernetes [1], Borg [2], and Twine [3]. These systems manage large-scale cluster resources and lifecycles of all applications running atop them, including provisioning, upgrades, autoscaling, and reconfigurations. Cloud infrastructure systems have been widely used as the control plane of modern clusters, datacenters, and clouds. For example, Kubernetes has become the cluster management solution for over 50,000 companies globally [4].

Cloud infrastructure systems are the bottom layer of the modern cloud computing world, and their reliability is paramount. Bugs in cloud infrastructure systems can affect all the upper layer applications and lead to disastrous consequences, such as service outages, data loss, resource leak, and security issues. For example, a bug in Kubernetes that violates a cluster identity guarantee (i.e., each pod in the cluster should have a unique name) impacts all the applications deployed on Kubernetes [5, 6]. Historic failure reports show that even mature cloud infrastructure systems constantly fail, and some failures have caused production incidents with catastrophic consequences [5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18].

Reliability of cloud infrastructure systems is notoriously hard. First, cloud infrastructure systems perform complex tasks such as resource allocation and application lifecycle management without having well-defined protocols or algorithms to reference. As a result, developers of cloud infrastructure systems tend to implement system-specific best effort heuristics without having a principled approach to reason about correctness. This also poses challenges for maintaining and evolving the implementation as it is hard to ensure that a code change preserves the original functionality. For example, the logic for managing storage volumes in Kubernetes is intentionally written in the very verbose "space shuttle style" to ensure each branch is explicitly accounted for, and developers are highly discouraged from simplifying the code [19]. Despite the effort, bugs were still reported in the code [20, 21, 22].

Second, cloud infrastructure systems run within complex and dynamic environments and face reliability challenges such as concurrency and failures. For example, concurrent system components might have races on their shared state and might crash at any point (due to kernel or hardware failures). In addition, cloud infrastructure systems lack programming support for reliability, such as transaction with atomicity and isolation guarantees. It is very challenging for developers to anticipate all corner cases at testing time to rule out bugs.

The current practice for improving the reliability of cloud infrastructure systems is insufficient. Developers have written numerous unit, integration, and end-to-end test cases. Chaos engineering tools [23, 24, 25, 26] have also been applied to test cloud infrastructure systems in failure scenarios. However, these tests rely on ad-hoc heuristics or random search to find bugs. They only cover a small portion of the entire execution space and cannot provide any strong correctness guarantee. In addition, manual testing and imprecise techniques like chaos engineering cannot reliably reproduce bugs with nondeterministic triggers (e.g., concurrency bugs), making failure diagnosis hard.

As cloud infrastructure systems have become the foundation of the cloud computing world, their reliability has become an emerging and pressing problem. This dissertation focuses on improving the reliability of cloud infrastructure systems in a principled way using verification and testing techniques.

1.1 PROBLEMS AND CHALLENGES

1.1.1 How to specify correctness of cloud infrastructure systems?

Formal specifications are mathematic formulas that define system correctness, and having a specification is the first step to verify system correctness. Formal specifications serve as an informative interface between different layers of systems and an unambiguous contract between system developers and users. However, cloud infrastructure systems, just like many other systems, do not have formal specifications.

Designing a good formal specification is a challenging task. First, the specification should capture the essential functionality of cloud infrastructure systems. Proving that the system meets the specification should preclude serious bugs that happen in real world.

Second, the specification should be general. Cloud infrastructure systems consist of a fleet of small components called controllers, and each controller manages one type of cluster resources or applications. The specification should be generally applicable to different controllers with diverse functionality.

Third, the specification should be realistic so it is possible to build a system that implements the specification. As an example, consider a specification stating that a fault-tolerant consensus protocol always eventually achieves consensus when running in any arbitrary environment—it is impossible to design such a consensus protocol (FLP impossibility) [27]. To make the specification realistic, we need to specify appropriate assumptions on the environment.

Finally, the specification should be concise enough for manual inspection. The value of formal verification lies not in providing an absolute guarantee of correctness, but in reducing the amount of code developers need to inspect—from thousands of lines of complex implementation to a concise specification, often just tens of lines. Specification bugs can undermine the verification effort [28, 29].

Given these challenges, this dissertation tries to answer the question: What should be the formal specification for cloud infrastructure systems?

1.1.2 How to build provably correct cloud infrastructure systems?

Formal verification offers very strong correctness guarantee that a system is mathematically proved to meet a specification for *all* possible system executions, including all possible inputs, event orderings, and failures. However, to achieve this guarantee, developers often have to manually write machine checked proof.

There has been a lot of progress in building formally verified systems, including operating systems [30, 31], compilers [32], file and storage systems [33, 34, 35, 36, 37, 38, 39, 40], and distributed systems [41, 42, 43]. For example, IronFleet [41] verified that a Paxos-based replicated state machine system offers linearizability, and a lease-based sharded key-value store behaves simply like a hash table.

Despite all the progress, writing proofs for system code still remains a highly challenging task, especially for cloud infrastructure systems. Recent research [37, 39, 42, 44] reports that the proof-to-code ratios are often larger than ten, meaning that for each line of implementation code, one needs to write ten or more lines of proof to verify the implementation. Cloud infrastructure systems have complex implementations with millions of lines of code, so modularization is the key to making verification possible. Since cloud infrastructure systems' core logics are modularized as controllers, we envision a practical verification approach that gradually verifies the entire cloud infrastructure system by incrementally replacing existing controllers with verified ones.

However, verifying controllers still poses many challenges. 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 these controllers run in a complex and dynamic environment, where they must handle unexpected faults, asynchrony, conflicts when interacting with other systems.

This dissertation explores how to build practical and formally verified controllers with manageable proof effort.

1.1.3 How to test existing cloud infrastructure systems comprehensively and efficiently?

Formal verification provides strong correctness guarantees, but we still need to test existing controllers continuously and extensively before there are verified replacements. In addition, verifying the controller code does not provide an end-to-end correctness guarantee for the entire system stack because lower layer systems, such as operating systems and compilers, are not verified yet. Testing helps discover bugs that originate from unverified code or the boundary between the verified and unverified.

Previous research has made great progress in testing distributed systems [45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62], but they do not offer a generally applicable, comprehensive, and efficient testing approach for diverse controllers. For example, existing fault-injection tools [45, 46, 47, 48, 49, 50, 51, 52] and concurrency-testing tools [53, 54, 55, 56] often (1) rely on an expert's hypotheses about vulnerable regions in the code under test, which makes it hard to apply to an ecosystem of diverse controllers, or (2) perform random testing, which makes it hard to test controllers comprehensively. Implementation-level model checking [57, 58, 59, 60, 61, 62] techniques can exhaustively explore a system's execution space in a principled way, but they suffer from the state explosion problem.

Testing controllers is a challenging task. First, controller bugs tend to have sophisticated triggering conditions. For example, a bug only happens when the order of a specific pair of events flips, or when a failure happens right between a pair of events. The testing tool should precisely capture diverse types of triggering conditions to detect bugs, and reliably replay triggering conditions to reproduce bugs. Second, controller bugs can lead to silent failures which are hard to observe by simply monitoring process status or logs. The testing tool should have oracles that catch even silent failures caused by controller bugs. Third, controllers have diverse implementations and features. To be generally applicable, the testing tool should not assume too much about controllers' implementation details.

Given these challenges, this dissertation explores how to develop a generally applicable, comprehensive, and efficient testing tool for controllers.

1.2 THESIS STATEMENT

Modeling controller behaviors as the cluster state's evolution enables (1) formal verification for building practical, provably correct controllers, and (2) testing for detecting serious, previously unknown controller bugs in an automatic, comprehensive, and efficient fashion.

1.3 CONTRIBUTIONS

To answer the questions in Section 1.1, this dissertation makes contributions to verification and testing techniques for cloud infrastructure systems. We first present state-centric reasoning, an approach for reasoning about controllers of cloud infrastructure systems. This reasoning approach models behaviors of diverse controllers in a uniform representation and enables verification and testing techniques that are generally applicable to diverse controllers. Based on state-centric reasoning, we present the first general controller correctness specification to address the problems from Section 1.1.1, the first framework for building practical and formally verified controllers to address the problems from Section 1.1.2, and the first automatic reliability testing tool for controllers to address the problems from Section 1.1.3.

1.3.1 State-centric Reasoning for Cloud Infrastructure Systems (Chapter 2)

We present *state-centric reasoning*, an approach for reasoning about behaviors of diverse controllers without knowing their implementation details. The key idea is to model each controller's behavior as a sequence of *cluster states*, instead of the controller's internal state.

State-centric reasoning is designed based on the state reconciliation principle followed by modern cloud infrastructure systems: Controllers implement the cluster management logic with a `Reconcile()` function that continuously monitors the actual cluster state and the desired state description and continuously updates the cluster state toward matching the desired state. State-centric reasoning leverages an important opportunity in this design: The cluster state is represented as a set of highly introspectable objects that share a uniform schema, and there is a clean separation between controllers and the state objects. The state objects affect and reflect controllers' behaviors, providing a vantage point to reason about controller correctness. This design makes state-centric reasoning generally applicable to diverse controllers.

State-centric reasoning models each controller's behavior as the cluster state's evolution, instead of the controller's internal state. This modeling allows state-centric reasoning to represent correct or buggy controller behaviors without revealing the controller's implementation details. We use a real-world example to illustrate how state-centric reasoning works.

The power of state-centric reasoning lies not only in reasoning but also in guiding the design of verification and testing techniques. State-centric reasoning enables (1) formal verification for controllers because asserting the cluster state's evolution is the natural way to formalize controller correctness, and (2) efficient and effective testing as it narrows down the testing space by focusing on events that affect the state reconciliation process.

1.3.2 Eventually Stable Reconciliation: Specifying Controller Correctness (Chapter 3)

The first step to formally verify a system is to define its formal specification. We present *eventually stable reconciliation (ESR)*, the first general formal specification of controller correctness. ESR is designed based on state-centric reasoning: ESR captures the essential functionality that controllers should provide by defining a set of correct cluster state sequences. Proving that a controller implementation meets the ESR specification shows that all possible behaviors of the controller are in the set.

The key insight behind ESR is to formalize controller correctness as a *liveness* property, instead of safety. Informally speaking, a liveness property states that something good *eventually* happens, while a safety property states that something bad *never* happens. Although most previous work [30, 31, 33, 34, 35, 36, 37, 38, 39, 40, 42, 43] focuses on specifying and verifying safety properties for other systems, we find that liveness naturally captures controller correctness in a general and concise form: A controller should eventually make the cluster state match a desired state. To be realistic, ESR also encodes a premise that the desired state eventually becomes stable, otherwise the controller might keep chasing a moving target. We formalize this liveness property in Temporal Logic of Actions (TLA) [63].

ESR precludes a broad range of controller bugs caused by factors like inopportune failures and conflicts with other controllers. Our analysis shows that ESR precludes 69% of all the bugs detected by state-of-the-art controller testing tools [64, 65].

Summary: ESR is the first general formal specification for controllers of cloud infrastructure systems, and it enables formal verification for cloud infrastructure systems. ESR precludes a broad range of bugs with diverse root causes and serious consequences, including the majority of the bugs detected by state-of-the-art controller testing tools.

1.3.3 Anvil: Building Formally Verified Controllers (Chapter 4)

A practical path toward fully verified cloud infrastructure systems is to replace existing, unverified controllers with verified controllers incrementally. To achieve this goal, we present Anvil, the first framework for implementing practical and formally verified controllers. Anvil allows developers to implement controllers in the Rust programming language and verify liveness and safety properties for the implementations. The verified controllers can be readily deployed in real-world cloud infrastructure systems.

Anvil emphasizes verifying both liveness and safety properties for practical implementations. Liveness verification is often harder than safety verification: Proving liveness often requires temporal logic reasoning for the entire system executions while proving safety invariants only requires induction on system transitions. To enable liveness verification for system implementations, Anvil's key idea is to combine Hoare-style [66] and TLA-style [63] verification. To verify liveness for a controller, developers structure their controller implementation as a state machine, prove that the controller implementation conforms to an abstract controller model using Hoare-style verification, and then prove liveness of the controller model using TLA-style verification.

Another 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 finite number of faults can happen, but eventually faults stop happening. This assumption is similar in spirit to partial synchrony [67] but for faults.

To reduce manual proof effort, Anvil provides verification support, including a reusable model for controller environments and lemmas encoding common reasoning patterns. To verify a controller, 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. 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.

Since ESR is a general specification, Anvil provides a general proof strategy for ESR that developers can follow. The key idea of the proof strategy is to divide the proof into two parts: Developers first prove that the environment eventually becomes stable (under the fairness assumptions), and then they prove that the controller makes progress toward the desired state starting from any possible cluster state in the stable environment.

Anvil leverages state-of-the-art automatic verifier. Specifically, Anvil is built on top of Verus [68, 69], an SMT-based deductive verification tool for Rust. Verus does not support temporal logic reasoning, so Anvil provides a TLA embedding on top of first-order logic to enable TLA-style temporal reasoning.

We also present case studies of using Anvil to implement and verify three practical Kubernetes controllers for managing ZooKeeper, RabbitMQ, and FluentBit. These controllers can readily be deployed in real-world Kubernetes platforms; they provide feature parity with respect to existing mature, widely used (but unverified) controllers. We have verified that the three controllers implement the ESR specification using Anvil. We have also verified a safety property specific to the RabbitMQ controller: the controller never performs unsafe scaling operations. 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.

Our evaluation shows that the verified controllers achieve competitive performance compared to unverified ones, and applying the state-of-the-art controller testing tools finds no bugs in the verified code.

Summary: Anvil is the first framework for building practical and formally verified controller implementations. With Anvil, we have built the first formally verified Kubernetes controllers. Anvil enables a practical path toward fully verified cloud infrastructure systems by replacing existing controllers with verified ones incrementally.

1.3.4 Sieve: Automatic Reliability Testing for Controllers (Chapter 5)

While Anvil allows developers to build verified new controllers, existing controllers still depend on manually written, ad-hoc test cases. We present Sieve, an automatic reliability testing tool that is generally applicable to controllers. Sieve's testing is guided by state-centric reasoning, and it achieves comprehensiveness, efficiency, usability, and reproducibility.

The key idea of Sieve is *state perturbation*, a testing approach that focuses on perturbing how a controller advances the cluster state's evolution. The insight is that a controller's action depends on its view of the cluster state. State perturbation tests a controller by perturbing the controller's view of the cluster state in ways it is expected to tolerate. It then compares the cluster state's evolution with and without perturbations to detect liveness and safety issues.

Sieve leverages an important opportunity in cloud infrastructure systems: Controllers interact with the cluster state objects via *state-centric interfaces* that perform semantically simple operations on the cluster state (e.g., reads and writes) and deliver notifications about cluster-state changes; the state objects that flow through the interfaces typically have a uniform schema. Thus, state-centric interfaces are an ideal vantage point to observe and perturb a controller's view of the cluster state. This makes Sieve generally applicable to diverse controllers.

Sieve performs comprehensive testing to detect different types of bugs. Sieve currently uses three different state perturbation patterns through injecting crashes, delays, and reconfigurations. These are circumstances that reliable controllers are expected to tolerate. For each pattern, Sieve automatically generates test plans that cover all possible perturbations during an execution of the controller under a given test workload.

Sieve automatically flags buggy behaviors using differential test oracles that compare the cluster state's evolutions with and without perturbations. The differential oracles allow Sieve to catch liveness and safety violations, including silent failures. The differential oracles are often more effective than searching for errors in logs and more comprehensive than human-written assertions.

Sieve's testing is efficient. Compared to techniques like implementation-level model checking that exhaustively manipulates message ordering and timing of failures, Sieve reduces the testing space by focusing only on the events that affect a controller's view of the cluster state. Sieve also avoids redundant and futile state perturbations to maximize test efficiency.

Sieve is highly usable. It does not require (1) formal specifications of the controller, (2) hypotheses about vulnerable regions in the code where bugs may lie, or (3) highly specialized test inputs. It does not rely on expert written assertions either. Sieve requires only a manifest for building the controller image and basic test workloads. Sieve's testing is then fully automatic. This degree of usability is key to making reliability testing broadly accessible to the rapidly increasing number of controllers.

Sieve reliably reproduces triggered bugs by replaying the state perturbation. For each state perturbation, Sieve generates a test plan that encodes what events to inject and when to inject to implement this state perturbation. Developers can replay the state perturbation by simply rerunning the corresponding test plan. For example, if a bug was triggered by injecting a crash right between two controller updates, Sieve can reproduce this bug by rerunning the same workload and injecting a crash between the same pair of events.

We evaluated Sieve on ten popular open-source controllers of various kinds, from either commercial vendors or official projects. Sieve found 46 new bugs in total, among which 35 have been confirmed (22 fixed) after we reported them. Notably, these are deep semantic bugs that Sieve detected without any expert guidance. The bugs have severe consequences, including application outages, security vulnerabilities, resource leaks, and data loss. Sieve is highly efficient—all controllers could be tested in under seven hours on a cluster of 11 machines, representing a typical nightly test. Sieve also has a very low false-positive rate of 3.5%, making its testing results trustworthy.

Summary: Sieve is the first automatic reliability testing tool for controllers. Sieve is designed to be generally applicable by testing at the right level—perturbing how the controller advances the cluster state's evolution. Sieve has improved the reliability of ten popular Kubernetes controllers by finding 46 serious new bugs, among which 22 have been fixed.

1.4 OUTLINE

The structure of this dissertation is as follows. Chapter 2 presents state-centric reasoning with background information about cloud infrastructure systems and controllers. Chapter 3 presents eventually stable reconciliation (ESR), the controller correctness formal specification. Chapter 4 presents Anvil, the framework for building and verifying controllers. Chapter 5 presents Sieve, the automatic reliability testing tool for controllers. Chapter 6 discusses related work on verification and testing. Chapter 7 concludes this dissertation and discusses future work. Appendix A gives a detailed example of how to verify liveness using Anvil.

The materials in some chapters have been published as conference papers. The materials in Chapter 3 and Chapter 4 have been presented in the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24) [70]. The materials in Chapter 5 have been presented in the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22) [64].

CHAPTER 2: STATE-CENTRIC REASONING FOR CONTROLLERS

Modern cloud infrastructure systems like Kubernetes [71], Borg [2], Twine [3], Omega [72], and vSphere [73] break down cluster-management logic into a fleet of microservices, called *controllers* [74]. For example, in Kubernetes, *all* the cluster-management logic is encoded in different controllers. Today, thousands of controllers are implemented by commercial vendors and open-source communities to extend Kubernetes with new capabilities [75, 76, 77, 78]. Controllers manage everything from application lifecycles (e.g., provisioning, upgrades, autoscaling) to stateful services, storage, networking, and integrations with cloud providers [79, 80, 81, 82, 83]. All controllers perform critical operations, making their reliability paramount.

To develop verification and testing techniques for controllers, our first step is to develop a general approach to reason about controllers' behaviors. We present *state-centric reasoning*, which represents a controller's behavior as the cluster state's evolution driven by the controller. State-centric reasoning provides a uniform representation of diverse controllers' behaviors without knowing their internals. This is achieved by leveraging an important opportunity in cloud infrastructure systems: There is a clean separation between controllers and the cluster state, and the cluster state is represented as highly introspectable objects. This reasoning capability enables verification and testing techniques that are generally applicable to diverse controllers and opens up opportunities to build fully verified cloud infrastructure systems in a practical manner.

The contribution of this chapter is state-centric reasoning, a general approach for reasoning about behaviors of controllers in cloud infrastructure systems. In this chapter, we first introduce the architecture and reliability challenges of cloud infrastructure systems, and then we present state-centric reasoning.

2.1 BACKGROUND

Cloud infrastructure systems are architected as a fleet of *controllers*. The controllers implement *all* logic for managing cluster resources, services, and applications. All controllers follow the same behavior pattern called *state reconciliation*, where controllers repeatedly attempt to reconcile between the current cluster state and some desired state. The cluster state is represented as mere structured data (e.g., JSON objects).

Cloud infrastructure systems rely on a clean separation between the cluster state and the controllers [74]. The structured data that represents the cluster state is stored in some logically centralized, highly-consistent datastore. The cluster state is exposed to controllers by an ensemble of API servers with state-centric interfaces that perform semantically simple operations on the cluster state (e.g., reads and writes).

We use Kubernetes as a representative example to present the basics of cloud infrastructure systems' architecture and the state-reconciliation principle. Figure 2.1 illustrates Kubernetes' architecture.

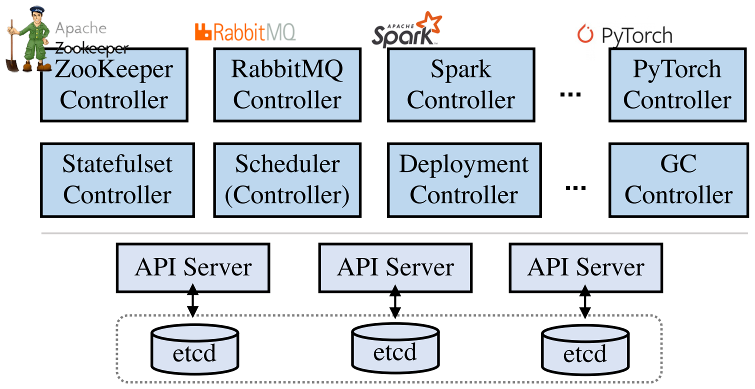


Figure 2.1: **Overview of Kubernetes.**

In Kubernetes, the cluster state is represented by shared data objects in etcd. Every entity in the cluster has a corresponding object in the cluster state, including pods, volumes, nodes, and groups of applications. These state objects are exposed by REST-based API servers and are stored in a logically centralized data store like etcd [84].

All Kubernetes' cluster-management logic is encoded in 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 [85, 86]. 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 toward 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. 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 2.2 shows a simple example of `Reconcile()`. The `Reconcile()` is invoked with a name pointing to the corresponding desired state description. During `Reconcile()`, the controller first gets the state object that encodes the desired state description, and then checks the current state of running pods. If the desired state requires more pods than currently exist, the controller will enter a loop to create new pods until the number matches.

```go
func Reconcile(dname string...) (err error) {
    desiredState, err := Get(dname)
    if err != nil { return err }
    pods, err := List("pods")
    if err != nil { return err }
    diff := desiredState.replicas - len(pods)
    if diff > 0 {
        for i := range diff {
            pod := make_new_pod(...)
            if err := Create(pod); err != nil {
                return err
            }
        }
    } else if diff < 0 {
        for i := range -diff {
            if err := Delete(pods[i]); err != nil {
                return err
            }
        }
    }
}
```

Figure 2.2: **A simplified example of `Reconcile()`.** This controller's goal is to keep the number of running pods the same as `replicas` from the desired state description. The controller interacts with state objects via `Get`, `List`, `Create`, `Delete`, and other APIs.

If the desired state requires fewer pods, the controller will delete pods iteratively. After the `Reconcile()` finishes, the number of actual running pods should match the desired state, assuming that no other processes creates or deletes pods. If the desired state or the running pods change again, `Reconcile()` will be invoked again to bring the cluster state to the (new) desired state.

This design makes Kubernetes extensible without requiring changes to its client library or interface—supporting a new system or feature is thus simply a matter of adding a custom controller and a corresponding set of custom object types to the cluster state. The design also allows controllers to be loosely coupled, which improves resilience: Controllers can independently fail and new controller instances can continue reconciliation.

Kubernetes' extensibility has led to the emergence of a thriving ecosystem of thousands of custom controllers that have been developed by commercial vendors and open-source communities [75, 76, 77, 79, 81, 82, 83]. For example, OpenShift, an enterprise Kubernetes platform from Red Hat, provides 160+ controllers that extend Kubernetes [87]. Amazon Web Service (AWS) provides 50+ controllers to manage AWS services using Kubernetes [88, 89].

Most cloud systems today have controllers to manage them atop Kubernetes. Even for the same cloud system, multiple controllers are developed to support different operation practices and deployment environments. Many critical cloud systems such as Istio [90], Crossplane [91], and StreamOps [92] are also implemented as controllers.

However, many real-world production incidents [5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18] show that developing correct and reliable controllers is challenging to achieve [93]. Popular, mature controllers misbehave when subjected to node crashes, network delays, and misoperations, causing system failures, service outages, data loss, etc.

To build a reliable controller, a developer faces the fundamental challenge of anticipating all possible cluster states that the controller needs to handle and safely reconciling to the required desired states from any of these points. There is a pressing need for verification and testing techniques that preclude controller bugs during the development phase.

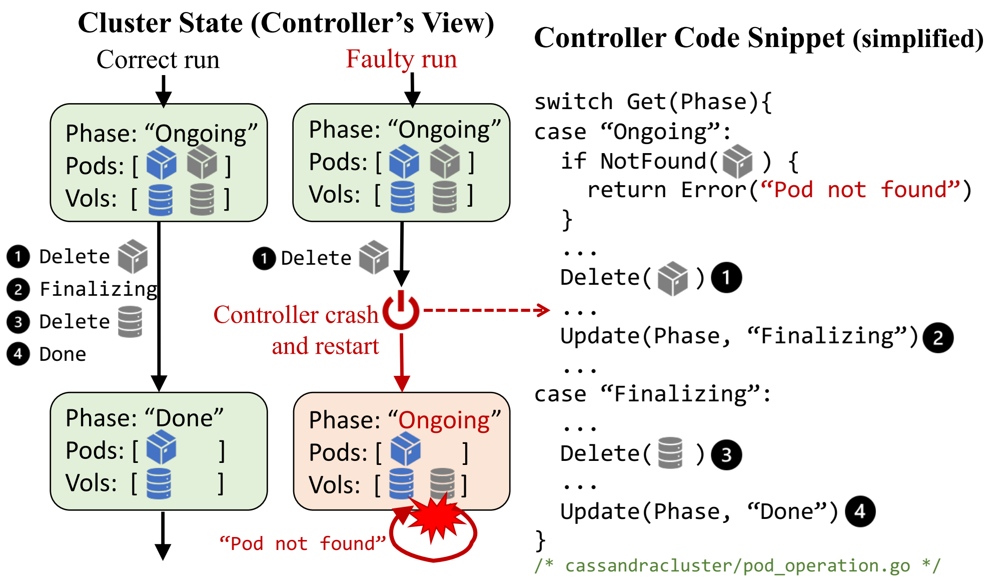
2.2 REASONING ABOUT THE CLUSTER STATE'S EVOLUTION

A key challenge of verifying or testing cloud infrastructure systems is how to reason about the behaviors of diverse controllers that manage different resources and applications. To address this challenge, we develop *state-centric reasoning*, which provides a uniform representation of state-reconciliation behaviors for diverse controllers.

State-centric reasoning leverages an important opportunity in cloud infrastructure systems: There is a clean separation between controllers and the objects that represent the cluster state. Although controllers have complex implementations and diverse features, the state objects share a uniform schema and enjoy high consistency provided by the underlying datastore (e.g., etcd). This provides us a vantage point to reason about the behaviors of diverse controllers without knowing their implementation details.

State centric reasoning represents a behavior of a controller as a sequence of cluster states, instead of the controller's internal state. State-centric reasoning focuses on how controllers advance the cluster state and abstracts away low-level controller implementation details. By observing the cluster state's evolution, we can differentiate correct controller behaviors from buggy behaviors.

As an example, Figure 2.3 shows one of many bug patterns of controllers [14, 64, 65, 93, 95]. The bug prevents the Cassandra cluster from auto-scaling and leaks storage resources (decommissioned volumes in gray are never deleted). This is because the controller lacks crash safety—it fails to recover from an intermediate state due to a crash between deleting a Cassandra pod and updating the `Finalizing` phase. With state-centric reasoning, the correct behavior of the controller is represented as a sequence of cluster states starting with



```
switch Get(Phase){
case "Ongoing":
  if NotFound( ) {
    return Error("Pod not found")
  }
  ...
  Delete( ) 1
  ...
  Update(Phase, "Finalizing") 2
  ...
case "Finalizing":
  ...
  Delete( ) 3
  ...
  Update(Phase, "Done") 4
}
/* cassandracluster/pod_operation.go */
```

Figure 2.3: **A bug in a Kubernetes controller for managing Cassandra [94].** The right side shows the buggy `Reconcile()` code, while the left side illustrates both correct and buggy executions of `Reconcile()` using state-centric reasoning.

two pods and two storage volumes and ending with one pod and one storage volume. The buggy behavior in the faulty run is represented as a sequence ending with an intermediate cluster state where the pod has been deleted but the volume still exists.

State-centric reasoning enables formal verification for controllers. The first step to formally verify a system is to formally specify its correctness guarantee. Based on state-centric reasoning, we design eventually stable reconciliation (ESR), a general formal specification for controller correctness (Chapter 3). ESR is made general by making assertions about the cluster state's evolution instead of controller implementation details. ESR precludes a broad range of controller bugs including the bug in Figure 2.3. We then develop Anvil, a framework for verifying controller implementations by reasoning about how controllers advance the cluster state (Chapter 4).

State-centric reasoning also enables efficient and effective testing for controllers. State-centric reasoning narrows down testing space by focusing on events that affect cluster state's evolution and provides a general testing oracle by comparing different cluster state evolutions. For example, the bug in Figure 2.3 was discovered by Sieve, a testing tool we developed based on state-centric reasoning (Chapter 5). Sieve analyzes the cluster state's evolution in the correct run of the Cassandra controller and then produces a faulty run where the controller starts its reconciliation process with an intermediate state (after a crash and restart). Sieve catches the triggered bug by comparing the end cluster states of the two runs. Sieve's testing is fully automatic with developer-provided testing workloads and does not require knowledge of controller-specific implementation details.

We present the verification and testing techniques designed based on state-centric reasoning in the following chapters.

CHAPTER 3: SPECIFYING CORRECTNESS FOR CONTROLLERS

Formal verification is a promising approach for guaranteeing correctness. To formally verify a program, developers need to write a formal specification that defines the correct executions of the program and a proof showing that the program meets the specification. This chapter focuses on the formal specification for controllers, and the next chapter presents how to prove that a controller meets the specification.

Designing a formal specification is challenging. First, the specification should capture the essential functionality of the system. Proving that the implementation meets the specification should preclude a broad range of bugs with serious consequences. Second, the specification should be generally applicable to diverse controllers for managing different resources and applications. Third, the specification should be realistic and practical. It should be possible for developers to write a system implementation that meets the specification. To be realistic, the specification should come with reasonable assumptions about the environment. Finally, the specification is a contract between system developers and users and there could be bugs in it, so we should make the specification simple and concise.

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).

ESR captures the essential functionality that controllers should provide, and it precludes a broad range of bugs caused by factors like inopportune failures and conflicts with other controllers. ESR is general and does not require controller-specific implementation details because it is designed based on state-centric reasoning and makes assertions about the cluster state's evolution, instead of a controller's internal state. ESR is also realistic and captures the necessary premise to reach the desired state. We formalize ESR as a concise formula in TLA (temporal logic of actions) [63], a linear-time temporal logic.

The contribution of this chapter is eventually stable reconciliation (ESR), the first general specification for controller correctness.

3.1 BACKGROUND ON TEMPORAL LOGIC OF ACTIONS (TLA)

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* [96]. We use TLA (temporal logic of actions) [63], a linear-time temporal logic well-suited to our needs.

TLA is a way of describing how state evolves over time. The semantics of TLA is about system executions, where an execution is an infinite sequence of states, including the current state and future states.

In temporal logics, each formula is interpreted as an assertion about executions. Each temporal formula is built up from elementary formulas using boolean operators (e.g., $\wedge$, $\vee$) and temporal operators (e.g., $\Box$, $\Diamond$). The temporal operators $\Box$ (always) and $\Diamond$ (eventually) are used in temporal logics to reason about future states. For example, if a formula F states that "Pod P exists in the current cluster state," then $\Box F$ states that "Pod P exists in the current cluster state *and all* future cluster states," and $\Diamond F$ states that "Pod P exists in the current cluster state *or some* future cluster state." Temporal logics such as TLA also allow nesting of temporal operators; for example, $\Diamond\Box F$ means that eventually we get to a point such that from that point onwards, F always holds.

We say that a temporal formula F is satisfied by an execution if F evaluates to true on this execution. A temporal formula F is said to be valid, written $\models F$, if and only if F is satisfied by all possible executions. Similarly, $F_2 \models F_1$ means that F_1 is satisfied by any execution that satisfies F_2. We often write `model` $\models$ `spec` to say that the `spec` is satisfied by all possible executions of the `model`.

3.2 EVENTUALLY STABLE RECONCILIATION (ESR)

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:

$$\forall d. \Box\big(\Box\texttt{desire}(d) \Rightarrow \Diamond\Box\texttt{match}(d)\big). \tag{3.1}$$

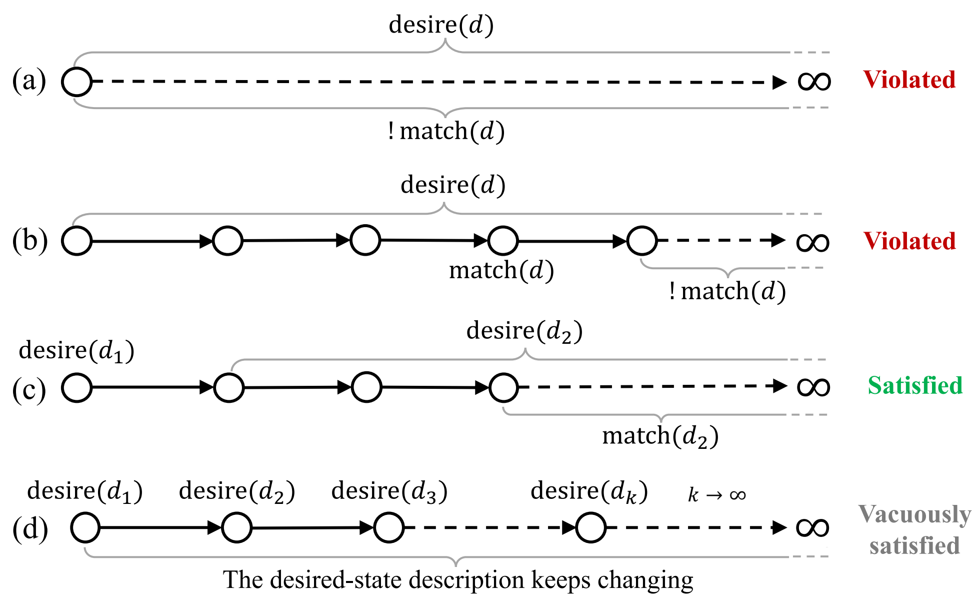


Figure 3.1: **Executions that violate or satisfy ESR.**

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.

ESR captures the key correctness properties shared by virtually all controllers: progress and stability. We elaborate on this with a detailed dissection of eq. (3.1). 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. (3.1) 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, $\forall d$ states that the controller reconciles all desired state descriptions.

Figure 3.1 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:

$$\texttt{model} \models \forall d.\, \Box\texttt{desire}(d) \leadsto \Box\texttt{match}(d). \tag{3.2}$$

3.3 PRACTICAL IMPLICATIONS OF ESR

Strictly speaking, ESR (eq. (3.1)) 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. (3.1), 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 [64, 65, 95]. For example, Sieve (Chapter 5) detected 28 bugs across ten popular controllers that the controller never matches the desired state due to inopportune failures and concurrency issues, which consist of 61% of all the bugs detected by Sieve. All such bugs are precluded by ESR. ESR also precludes 75% of all the bugs detected by Acto [65], a testing tool developed by us for custom application controllers by mutating desired state descriptions. Prior work [95] 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.

CHAPTER 4: BUILDING FORMALLY VERIFIED CONTROLLERS

To close the gap between formal specifications and practical controller implementations, we present Anvil, a framework for implementing practical controllers and formally verifying the implementations. We have developed and verified practical Kubernetes controllers for managing critical systems using Anvil. Anvil emphasizes verifying both liveness (e.g., ESR) and safety properties for system implementation code. The key idea of Anvil is to combine Hoare-style [66] and TLA-style [63] verification to first connect a controller implementation to an abstract state machine model, and then prove that all possible executions of the model satisfy liveness or safety properties.

A common challenge in proving liveness properties 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 finite number of faults can happen, but eventually faults stop happening. This assumption is similar in spirit to partial synchrony [67] but for faults.

To verify a controller implementation, 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 (Section 4.2.3). 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 (Section 4.2.4).

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 (Section 4.3), 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.

We implemented Anvil for verifying Kubernetes controllers on top of Verus [68, 69], 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 (Section 4.2.2) to enable TLA-style temporal reasoning.

We used Anvil to implement in Rust three practical Kubernetes controllers for managing ZooKeeper, RabbitMQ, and FluentBit (Section 4.4). 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.

In this chapter, we first present how to write controller implementations using Anvil. Then we present Anvil's verification support and how to prove ESR using Anvil. Finally we present case studies of building verified controllers using Anvil and evaluation.

Anvil's TLA-style verification support for proving liveness properties is applicable to any system. We demonstrate how to write TLA-style liveness proof using a small yet representative example in Appendix A.

Summary. This chapter makes the following contributions:

- We present Anvil, a framework for developing practical controllers and formally verifying that the controller implementations satisfy correctness properties such as ESR.
- We have built three representative and practical Kubernetes controllers and verified their correctness using Anvil.
- We present an evaluation of the end-to-end correctness and performance of the three verified controllers.
- We have made Anvil and the verified controllers publicly available at `https://github.com/anvil-verifier/anvil`.

4.1 IMPLEMENTING CONTROLLERS USING ANVIL

In Anvil, developers implement a controller using a state machine; this style is common practice in unverified controllers as well [97, 98], and in Anvil it enables TLA-style verification. Figure 4.1 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 4.2: it starts from the initial state and invokes `step()` iteratively until all steps are done or if any step encounters an error. Each iteration

```rust
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 4.1: **Anvil's basic Controller API.** To implement a controller, developers implement the Controller trait.

of `step()` returns the next 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 (Section 4.4.1). 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 [99]) which invokes `reconcile()` upon changes, handles its output, and requeues the next invocation.

Figure 4.3 shows the `step()` implementation of a controller that manages ZooKeeper [100, 101] on Kubernetes. 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 [102] from the Kubernetes API (`service_get_req`) and the next state to transition to `ReconcileService`. The `reconcile` method (Figure 4.2) fetches the service object using the

```
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 4.2: **Anvil code that assembles `reconcile()` using the Controller API in Figure 4.1.**

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`.

4.2 ANVIL'S VERIFICATION SUPPORT

Anvil is built on top of Verus [68, 69], an SMT-based deductive verification tool for Rust backed by Z3 [103], in similar spirit to Dafny [104]; it offers a Hoare-logic [66] framework for reasoning modularly about imperative code in Rust. To enable TLA reasoning, Anvil requires developers to implement their controllers as state machines.

Figure 4.4 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 4.3) 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. (3.2)).

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

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

Floyd-Hoare style reasoning (Section 4.2.1). 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 (Section 4.2.2) 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 (Section 4.2.3), 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 (Section 4.2.4) 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.

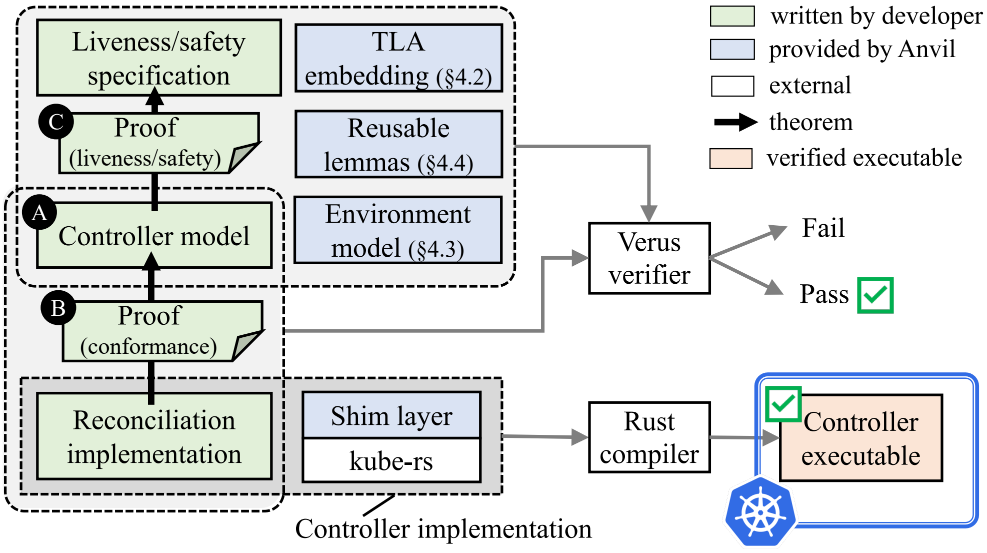


Figure 4.4: **An overview of Anvil's workflow.**

Assumptions. Anvil relies on the following assumptions: (1) The TLA embedding correctly defines TLA concepts [63]. (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.

4.2.1 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 [38, 41]. 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 4.5. This API defines a state machine and is similar to the Controller API in Figure 4.1, except that all the methods and variables are written in *ghost code* [68, 104]. 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 4.5: **Anvil's ControllerModel API.** Developers use the API to write the controller model (an abstract state machine). It mirrors the implementation trait (Figure 4.1) 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 4.1), developers write a corresponding `m_step()` using the ControllerModel API (Figure 4.5). If the implementation's `step()` returns a Kubernetes-API request, `m_step()` correspondingly returns a ghost-type request (`ReqV`) that queries the Kubernetes API model (Section 4.2.3). 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 4.1), 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 4.6 shows the theorem to prove conformance for the ZooKeeper controller's `step()` in Figure 4.3. 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()`.

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 [99] library, but Verus cannot directly reason about definitions from unverified libraries. So,

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

Figure 4.6: **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.

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 4.6, 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 [66] 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.

4.2.2 TLA Embedding

To enable liveness reasoning on top of Verus, Anvil develops a TLA embedding that models important concepts in TLA. Anvil follows IronFleet [41] 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. (3.2)) as in Figure 4.7. 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 4.3

checks if the service, config map and stateful set exist in the data store and match the desired state description `d` (Figure 4.8).

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

Figure 4.7: **The ESR theorem specified using the TLA embedding.**

```
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 4.8: **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`).

In the style of specifying systems [105], 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 predicates* over individual states or *action predicates* over individual transitions. Developers can convert a state predicate to a temporal predicate using a `lift` function [105]: 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 4.8. 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 (Section 4.2.4). This differs from IronFleet which interacts directly with instantiated executions throughout the liveness proof. We present Anvil's temporal reasoning style in Section 4.3.2 and a more detailed example in Appendix A.

4.2.3 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.

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 [84]) 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 [106] which deletes a state object if all of its listed owners have been deleted, (2) the StatefulSet controller [107] which manages stateful applications, and (3) the DaemonSet controller [108] which manages daemons processes (e.g., for monitoring).

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.

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* [63] 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 finite number of times but eventually stop happening, in the spirit of partial synchrony [67].

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 [109]. 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.

4.2.4 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

```
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 4.9: **The leads-to transitivity lemma.**

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.

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 4.9). 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 lemmas, representing a broad range of temporal reasoning patterns.

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 (Section 4.2.3). The GC's job is to delete orphan objects whose owner [110] 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 [111], 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 4.10). 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 Section 4.4.

```
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 4.10: **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.

4.3 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.

4.3.1 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 becomes 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 4.11 shows the high-level proof structure.

```
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 4.11: **High-level structure of the ESR proof.** `model` describes the original environment in Section 4.2.3. `stable_model`(d) describes the stable environment: faults and conflicts stop, and the desired state d is stable.

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 (Section 4.2.3).

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 Anvil's lemma on the GC (Figure 4.10) 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 stabilizes, any controller request for any previous version of the desired state eventually leaves the network.

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 4.3, 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 4.12). First, we prove a termination property: the controller's current reconciliation (the current invocation of `reconcile()` in Figure 4.2) 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 4.2). 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 4.3) to prove that the controller eventually creates and updates all the state objects to match the desired state.

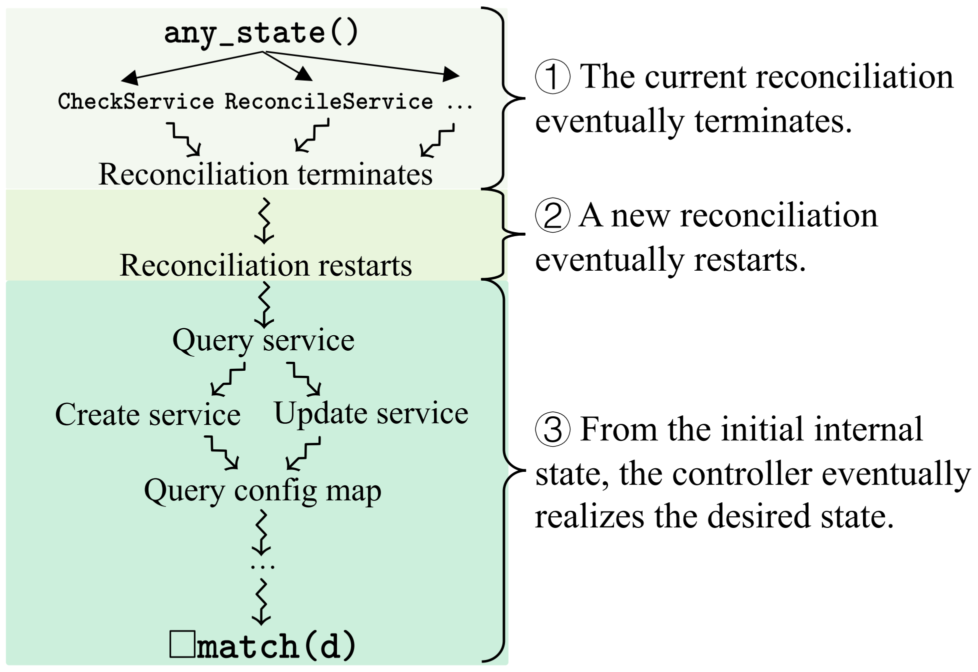

any_state()
CheckService
ReconcileService
...
Reconciliation terminates
Reconciliation restarts
Query service
Create service
Update service
Query config map
...
□match(d)
① The current reconciliation eventually terminates.
② A new reconciliation eventually restarts.
③ From the initial internal state,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

Figure 4.12: **Proving liveness in a stable environment.**

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 4.3 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.

4.3.2 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 4.13 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 4.13: **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 [63]. 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`.

4.4 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 [112, 113] 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 [114]. 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 4.1) 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 (Section 4.3). 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 [64, 65].

4.4.1 ZooKeeper Controller

We implement and verify a full-fledged ZooKeeper controller, using the controller [112] from Pravega [115] as the reference. Figure 4.3 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 4.2); 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 [116]. 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 reconfiguration, in ESR, `match` asserts that each state object matches the desired state description (as in Figure 4.8), 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 2.3. This led us to find a similar bug in the reference controller we reported in [117]. Recent work [64] 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 (Section 4.2.3), which helped us find this bug.

4.4.2 RabbitMQ Controller

We implement and verify a full-fledged controller for RabbitMQ, a widely used message broker [118]. We use the official RabbitMQ controller as the reference [113].

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 [119]. 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 [120]. 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.

4.4.3 FluentBit Controller

We implement and verify a controller for FluentBit, a popular logging and metrics service [121]. 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 [114] 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 (Section 4.5.1).

4.5 EVALUATION

We evaluate Anvil along the dimensions of verification effort (Section 4.5.1), controller correctness (Section 4.5.2) and performance (Section 4.5.3). Our evaluation shows that it is pragmatic to implement, verify and evolve practical Kubernetes controllers with Anvil.

4.5.1 Verification Effort

Table 4.1 shows the details of each verified controller we built using Anvil. The proof effort is manageable. 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 (Section 4.2.4) and proof strategy (Section 4.3). 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 [122, 123, 124, 125, 126, 127, 128, 129]. All our temporal logic reasoning is done by applying Anvil's temporal logic lemmas without unfolding the definition of executions and temporal logic operators.

	Trusted	**Exec**	**Proof**	**Time to Verify**
	(lines of source code)			(seconds)
ZooKeeper controller Section 4.4.1				
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 Section 4.4.2				
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 Section 4.4.3				
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 4.1: **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).

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 (Section 4.2.1). 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.

Verifying each controller takes under 3 minutes in real time on a 6-core 16 GB laptop with 11 parallel threads. 87% of proof functions verify in under ten CPU seconds, and the slowest of them takes 120 CPU seconds.

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 (Section 4.2.1). All the lemmas are verified in under one minute.

4.5.2 Controller Correctness

We run extensive end-to-end functional tests on the verified controllers using Acto [65]. 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 [64] for Rust controllers; we present Sieve in Chapter 5.

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

Table 4.2 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 [130]), 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.

4.5.3 Controller Performance

The verified controllers have comparable performance to the reference controllers. We use Acto [65] 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, 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.

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

Table 4.3 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 [3, 131]. 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.

4.6 LIMITATIONS

Formal verification is often a tradeoff between human efforts users need to pay and the correctness guarantee users get. Anvil provides strong correctness guarantee, but requires manual proof effort. Developers need to perform Hoare-style and TLA-style reasoning and write machine-checked proofs to finish verification. As the controller implementation evolves, developers also need to maintain and repair the proof. From our evaluation, we found the proof effort is manageable (proof-to-code line ratio ranges from 4.5 to 7.4).

Anvil cannot be directly applied to verify arbitrary controller implementations. To apply TLA-style reasoning, Anvil requires developers to structure a controller implementation as a state machine, and each action of the state machine must be atomic with respect to cluster-state changes. From our experience, the state machine model is general and expressive enough to write controller implementations.

4.7 DISCUSSION

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 (Section 4.5.2). 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 Section 4.4.2 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 2.3 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?) [60]. 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 (Section 4.2.3), 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.

4.8 SUMMARY

In this chapter, we present Anvil, a framework for developing and verifying controllers. With Anvil, developers write controllers in Rust and verify safety and liveness properties, including ESR, using TLA-style deductive verification. 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 toward provably correct cloud infrastructures. We have made Anvil publicly available at `https://github.com/anvil-verifier/anvil`.

CHAPTER 5: RELIABILITY TESTING FOR CONTROLLERS

Formal verification provides strong correctness guarantees, but we still need to test existing controllers continuously and extensively before there are verified replacements. In addition, verifying the controller code does not provide an end-to-end correctness guarantee for the entire system stack because lower layer systems, such as operating systems and compilers, are not verified yet. Testing helps discover bugs that originate from unverified code or the boundary between the verified and unverified.

The existing testing practices fall short. Many popular controllers adopt mature software testing practices and have numerous unit, integration, and end-to-end test cases. Some even test scenarios involving faults. However, manually-written test suites do not sufficiently test a controller's reliability as it is prohibitively difficult for developers to anticipate all possible cluster states, let alone codify them into test cases.

In this chapter, we present Sieve, an automatic reliability testing tool for controllers. The key idea of Sieve is state perturbation, a general testing approach for state-reconciliation systems based on state-centric reasoning. Sieve perturbs the controller's view of the cluster state in ways it is expected to tolerate, and then compares the cluster state's evolution with and without perturbations to detect triggered bugs.

Sieve's testing covers diverse types of bugs. Sieve tests a controller by exhaustively introducing state perturbations through crashes, delays, and reconfigurations. These are circumstances that reliable controllers are expected to tolerate. Currently, Sieve supports three typical perturbation patterns that expose controllers to (1) intermediate states (Figure 2.3), (2) stale states (or past cluster states), and (3) unobserved states due to missing some cluster state transitions (Section 5.2.1).

For each pattern, Sieve automatically generates test plans that cover all possible perturbations during an execution of the controller under test. Test-plan generation is based on analyzing a controller's behavior and the cluster-state evolution during reference executions. Sieve effectively avoids redundant and futile test plans to maximize test efficiency.

Sieve automatically detects buggy controller behavior using differential test oracles that compare the cluster-state transitions with and without perturbations. This comparison is feasible because a controller's behavior is reflected in the sequence of cluster-state transitions. The differential oracles are often more effective than searching for errors in logs and more comprehensive than human-written assertions (Section 5.2.6).

Sieve is highly usable. Sieve does not require (1) formal specifications of the controller or the cloud infrastructure system, (2) hypotheses about vulnerable regions in the code where bugs may lie, or (3) highly specialized test inputs. It does not rely on expert written assertions either. Sieve requires only a manifest for building the controller image and basic test workloads. Sieve's testing is then fully automatic. This degree of usability is key to making reliability testing broadly accessible to the rapidly increasing number of controllers.

We evaluated Sieve on ten popular open-source controllers for managing critical cloud systems (e.g., ZooKeeper, MongoDB, Cassandra). The controllers are from either commercial vendors or official projects of the managed applications. Sieve found 46 new bugs in total, among which 35 have been confirmed (22 fixed) after we reported them. Notably, these are deep semantic bugs that Sieve detected without any expert guidance. The bugs have severe consequences, including application outages, security vulnerabilities, resource leaks, and data loss. Sieve is highly efficient—all controllers could be tested in under seven hours on a cluster of 11 machines, representing a typical nightly test. Sieve also has a very low false-positive rate of 3.5%, making its testing results trustworthy.

Summary. The chapter makes four main contributions:

- We present state perturbation, the first automatic reliability-testing technique for state-reconciliation systems: exhaustively perturbing the controller's view of cluster states and using differential oracles on the cluster state evolution to detect bugs.
- We design and implement Sieve, a system that uses our proposed technique to automatically test *unmodified* cluster-management controllers in Kubernetes.
- Sieve has already improved the reliability for ten popular open-source controllers by virtue of bugs it found that were then fixed by developers. It is practical to run Sieve regularly.
- We have made Sieve publicly available at `https://github.com/sieve-project/sieve`, with instructions to reproduce all discovered bugs.

5.1 STATE PERTURBATION

We present state perturbation, a comprehensive and efficient testing approach for controllers designed based on state-centric reasoning. The key idea of state perturbation is to perturb a controller's view of the cluster state in ways the controller is expected to tolerate, and then compare the cluster state's evolution with and without perturbations to automatically detect safety and liveness issues.

State perturbation is powered by a fundamental opportunity in cloud infrastructure systems—the cluster state is represented by objects, and controllers interact with the cluster state objects via *state-centric interfaces* (in Figure 5.1). State-centric interfaces perform semantically simple operations on the cluster state (e.g., reads and writes) and deliver notifications

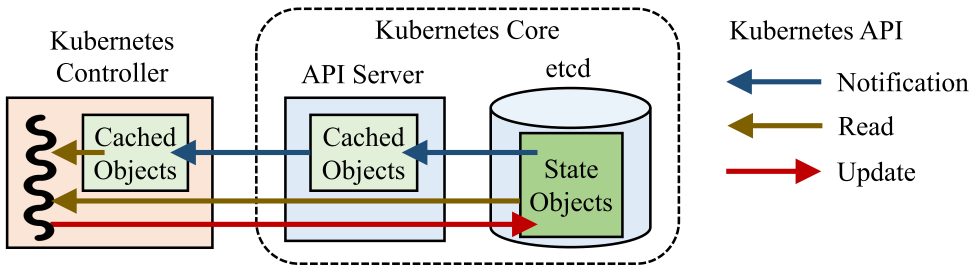


Figure 5.1: **Interaction between a Kubernetes controller and the state objects.** Any update to the state objects stored in etcd is propagated to a local cache of each API server and subsequently to the controller's local cache. The controller reads each state object either from its local cache (for performance) or via a quorum read from etcd (for consistency).

about cluster-state changes; the objects that flow through the interfaces typically have a uniform schema. State-centric interfaces are highly introspectable and hence an ideal vantage point to observe and perturb a controller's view of the cluster state.

State perturbation can be implemented by injecting events that controllers should tolerate, such as node crash or time delay. Compared to directly reasoning about when to inject what events, state perturbation narrows down the testing space by focusing on the events that effectively lead to different state reconciliation decisions made by controllers.

5.2 SIEVE'S DESIGN

Sieve checks whether the controllers under test can correctly operate the system under common perturbations (due to unexpected faults and inherent asynchrony) and detects bugs that lead to safety and liveness issues at the development time. Sieve is automatic—it tests unmodified controllers and does not rely on formal specifications or controller-specific assertions. Sieve is effective—it focuses on well-defined, highly-targeted perturbations that reliable implementations are required to tolerate.

Sieve perturbs the controller's view of the cluster state based on three broad patterns that expose the controller to (1) intermediate states, (2) stale states, and (3) unobserved states. We discuss the three patterns and their rationales in Section 5.2.1. Note that these are not the only patterns in which faults can occur, but cover a broad range of faults that a component in a distributed system is expected to handle gracefully. Sieve can be extended to incorporate other patterns in the future.

Sieve tests controllers with the following workflow:

- *Collecting reference traces* (Section 5.2.2). Sieve starts by learning how a controller behaves in the absence of faults (under test workloads) and records the state transitions in reference traces. To do so, it instruments the state-centric interfaces used by the controller to interact with the cluster state.

- *Generating test plans* (Section 5.2.3). Sieve then analyzes the reference traces to generate *test plans*. A test plan describes a concrete perturbation. The test plan specifies *what* faults to inject and *when* to inject them to effectively drive the controller to see a target cluster state.
- *Avoiding ineffective test plans* (Section 5.2.4). To achieve high test efficiency, Sieve prunes redundant or futile test plans. For example, it avoids a test plan if it is clear that it cannot causally lead to a target cluster state.
- *Executing test plans* (Section 5.2.5). Sieve executes each test plan using a test coordinator. The test coordinator monitors the cluster-state transitions during testing and injects the specified faults according to the test plan's specification.
- *Checking test results* (Section 5.2.6). Sieve has generic, effective, differential oracles to automatically check test results. The oracles detect buggy controller behavior by comparing the cluster-state evolution between the reference and test runs.

Sieve deals with non-deterministic elements of the cluster state during testing to minimize their impact on test plan generation and test oracles (Section 5.2.7). Specifically, Sieve identifies non-deterministic state objects and fields and excludes them.

Usage. To use Sieve, one needs to provide two inputs: (1) a manifest that specifies how to build and deploy the controller under test, and (2) a set of test workloads that exercise end-to-end behavior of the controller under test. The two inputs are mostly available in mature controller projects, as they are needed for controller development and deployment. In our experience, finding them is straightforward.

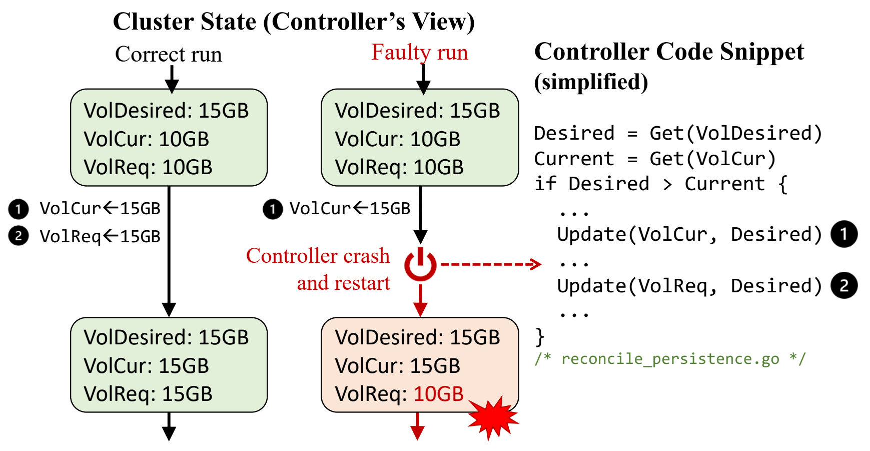
5.2.1 Perturbing a Controller's View of the Cluster State

Sieve operates under the assumption that a controller follows the state-reconciliation principle, which receives a sequence of *notifications* about the changes to the cluster states and outputs a corresponding sequence of *updates* to the cluster states. Sieve aims to affect the outputs of a controller by perturbing its view of the cluster state. These perturbations are produced by injecting targeted faults (e.g., crashes, delays, and connection changes) when specific cluster-state changes (*triggering conditions*) happen.

Notably, the perturbation strategy allows Sieve to *decouple policy from mechanism*. The decoupling makes it easy to extend existing policies or add new policies by orchestrating the underlying perturbation mechanisms. Specifically, a policy defines a view Sieve exposes to the controller at a particular condition, while the mechanism specifies how to inject faults to create the view. Sieve automatically generates test plans for each policy; each test plan introduces a concrete perturbation based on a specification of a triggering condition and a fault to inject when that condition happens.

Sieve currently supports three patterns to perturb a controller's view. Crucially, these perturbations drive a controller to states that it is expected to tolerate. They represent valid inconsistencies in the view that a controller could see due to common faults as well as the inherent asynchrony of the overall distributed system. Over time, we hope to add more perturbation patterns.

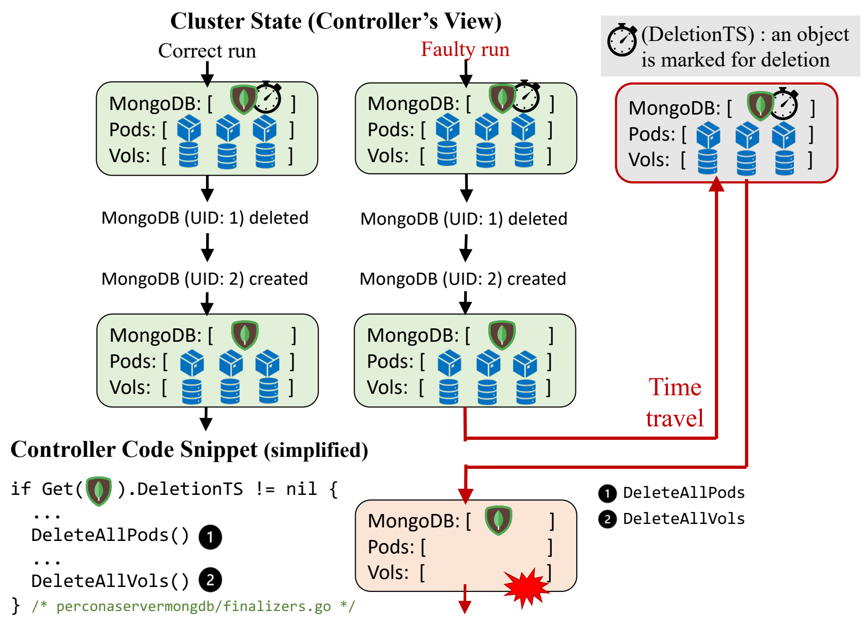
Intermediate states. Intermediate states occur when controllers fail in the middle of a reconciliation before finishing all the state updates they would have otherwise issued. After recovery (e.g., Kubernetes automatically starts a new instance of a crashed controller), the controller needs to resume reconciliation from the intermediate state left behind.



```
Desired = Get(VolDesired)
Current = Get(VolCur)
if Desired > Current {
  ...
  Update(VolCur, Desired) ❶
  ...
  Update(VolReq, Desired) ❷
  ...
}
/* reconcile_persistence.go */
```

Figure 5.2: **An intermediate-state bug in a RabbitMQ controller detected by Sieve [132].** The controller fails to recover from the intermediate state introduced by Sieve; the controller does not successfully resize the storage volume.

Figure 5.2 illustrates how Sieve tests the official RabbitMQ controller with intermediate-state perturbations and reveals a new bug. The test workload attempts to resize the storage volume from 10GB to 15GB. The resizing is implemented with two updates: (1) updating `VolCur` to 15GB; (2) updating `VolReq` to 15GB which triggers Kubernetes to resize the volume. The controller issues updates when `VolCur` is smaller than the desired volume size. During testing, Sieve crashes the controller between the two updates, which creates an intermediate state where `VolCur` is updated, but `VolReq` is not. The controller cannot recover from the intermediate state and the resizing never succeeds. The bug has been fixed with 700+ lines of Go code to revamp the volume resizing logic. In addition, the developers



```
if Get(   ).DeletionTS != nil {
  ...
  DeleteAllPods() 1
  ...
  DeleteAllVols() 2
} /* perconaservermongodb/finalizers.go */
```

Figure 5.3: **A stale-state bug in a MongoDB controller detected by Sieve [135].** The controller experiences a "time-travel" and observes a stale state. It makes wrong reconciliation action based on the stale state (deleting all the pods and volumes) which leads to application outages and data loss.

added eight new tests along with the fix to exercise how the controller handles different intermediate states, which is what Sieve performs automatically.

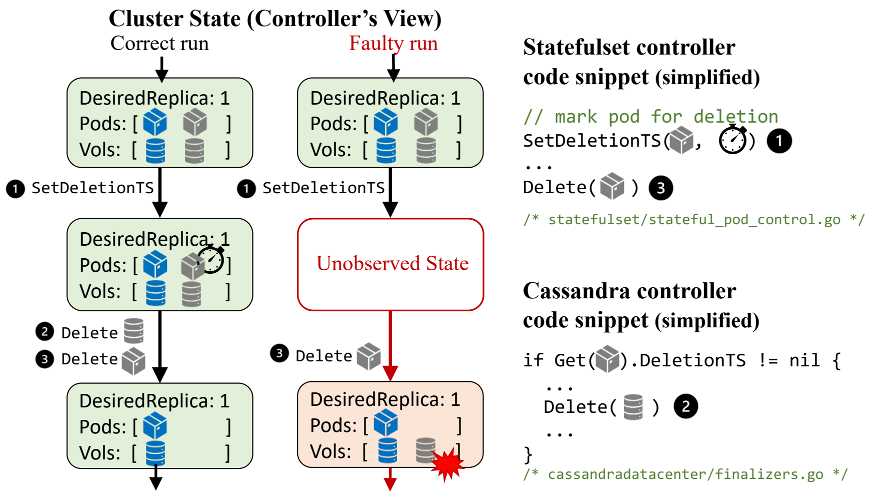
Stale states. Controllers often operate on stale states, due to asynchrony and the extensive uses of caches for performance and scalability [133]. As shown in Figure 5.1, controllers do not directly interact with the strongly consistent data stores, but are connected with API servers. The states cached at API servers could be stale due to delayed notifications. Controllers are expected to tolerate stale views that lag behind the latest states maintained in the data store.

Tolerating stale views correctly is nontrivial. For example, a Kubernetes controller's view may "time travel" to a state it observed in the past. Time traveling occurs when there are multiple API servers operating in a high-availability setup, when the controller reconnects to a stale API server that has not yet seen some updates to the cluster state. The reconnection can be triggered by failover, load balancing, or reconfigurations. Controllers are expected to recognize the stale state [134], instead of treating it as a new, unseen state.

Figure 5.3 illustrates how Sieve tests Percona's MongoDB controller with stale-state perturbation and reveals a new bug that leads to both application outages and data loss. To support graceful MongoDB cluster shutdowns, the controller waits to see a non-nil deletion timestamp (`DeletionTS`) field attached to the state object representing the MongoDB cluster (a common practice to give systems time to react to an impending deletion [136]). When the controller sees this change, it deletes all the pods and volumes of the MongoDB cluster.

Sieve drives the controller to mistakenly delete a live MongoDB cluster by introducing a time-travel perturbation. With a workload that first shuts down a MongoDB cluster and then recreates a new instance of the same cluster, Sieve waits till the cluster is recreated and then introduces a time-travel perturbation. The perturbation causes the controller to see the deletion timestamp being applied to the *already-deleted cluster*. Consequently, the controller mistakenly shuts down the newly created cluster. This revealed that the controller should be checking for the UIDs of clusters, not just their names.

Unobserved states. By design, controllers may not observe every cluster-state change in the system. The full history of changes made to the cluster state is prohibitively expensive to maintain and expose to clients [111]. Controllers are hence expected to be designed as *level-triggered* systems (opposed to being *edge-triggered*), i.e., a controller's decision must be based on the currently observable cluster state (level) [137], not on seeing every single change to the cluster state (edge).


Cluster State (Controller's View)
Correct run
Faulty run
DesiredReplica: 1
Pods: [ ]
Vols: [ ]
1 SetDeletionTS
2 Delete
3 Delete
Unobserved State
Statefulset controller code snippet (simplified)
Cassandra controller code snippet (simplified)

```
// mark pod for deletion
SetDeletionTS( , ) 1
...
Delete( ) 3
/* statefulset/stateful_pod_control.go */
```

```
if Get( ).DeletionTS != nil {
  ...
  Delete( ) 2
  ...
}
/* cassandradatacenter/finalizers.go */
```

Figure 5.4: **An unobserved-state bug in a Cassandra controller detected by Sieve [138].** The controller misses a transient state where the pod has a non-nil deletion timestamp. It thus fails to delete the volumes, leaking storage resources. The bug also prevents new Cassandra pods from rejoining.

Figure 5.4 illustrates how Sieve tests Instaclustr's Cassandra controller using unobserved-state perturbations and reveals a new bug that leads to resource leaks and service failures. The test workload first scales down and then scales up storage volumes of the Cassandra cluster. During scale-down, the controller removes volumes when it learns that the corresponding pods were marked for deletion (a non-nil deletion timestamp field is set on the pod object, similar to the previous example). The pods' lifecycles (including deletions) are managed by a built-in controller called a StatefulSet controller. Sieve pauses notifications to the Cassandra controller for a window such that it does not see these deletion marking events by the StatefulSet controller. This causes the Cassandra controller to not delete the corresponding volumes even though it has the right information to make that call (i.e., its view has volumes created by it that do not have pods attached to them).

Hence, the volume never gets deleted, leaking the storage resource. The bug also prevents the controller from scaling the Cassandra cluster – newly-created pods try to reuse the dangling volumes and cannot rejoin using the cluster metadata already in them (as it represents a node that was decommissioned). The bug has been fixed by adding a pre-deletion hook – a coordination mechanism in Kubernetes that allows the Cassandra controller to complete the required cleanup operations before the pods can be deleted [139].

5.2.2 Collecting Reference Traces

Sieve starts by learning how a controller behaves in the absence of faults. To do so, Sieve interposes around the state-centric interfaces used by the controllers to interact with the cluster state. All modern cloud infrastructure systems have unified, well-defined client libraries based on state-centric interfaces. Taking Kubernetes as an example, any interaction with the cluster state (exposed by the API servers) goes through a small, well-defined set of client APIs that read, modify, or receive notifications about state objects. They are used by every controller that interacts with the Kubernetes API servers. To support Kubernetes controllers, Sieve decorates 10 functions in the client library and this interposition is fully automated (Section 5.3).

With the interposition in place, Sieve learns every cluster-state change notification that the controller receives, as well as any reads and writes attempted by the controller to the cluster state or to the local cache of the cluster state. Sieve then runs each test workload supplied by the developer and collects the following two reference traces:

- *Controller trace.* A series of events observed via the interposition of client APIs, including notifications about state changes, entry and exits of each reconciliation cycle, and client-API invocation by the controller and their arguments.
- *Cluster state trace.* The initial cluster state and the sequence of state changes, collected using public APIs of the cloud infrastructure system.

The controller trace is used for generating test plans (Section 5.2.3) and the cluster-state trace is used by test oracles (Section 5.2.6).

5.2.3 Test Plan Generation

Sieve generates a set of test plans for each test workload for which it has collected reference traces. Each test plan specifies a perturbation to inject during the workload.

A test plan is represented by a self-contained file that describes a test workload, a list of faults to inject during the workload run, and the triggering condition for when to inject each fault. Sieve currently supports several primitives that test plans can compose to introduce complex faults: (1) crash/restart a controller, (2) disconnect/reconnect a controller to an API server, (3) block/unblock a controller from processing events, and (4) block/unblock an API server from processing events. When an executed test plan reveals a bug, the test-plan file is sufficient to reproduce the bug.

Figure 5.5 shows a simplified test-plan file generated by Sieve. Each element in `faults` specifies the fault to inject (`faultType`) and the triggering conditions (`triggers`). Each element in `triggers` specifies a triggering condition, that causes the specified fault to be injected before or after a particular cluster state change if executed. A composite triggering condition can be specified in `compositeTrigger` by combining multiple conditions in `triggers` with boolean operators. For example, `t1 & t2` means the fault is only injected when both `t1` and `t2` are triggered. In Figure 5.5, `trigger1` is the only required condition to inject the fault. Similarly, composite faults can be constructed (e.g. crashing a controller after `t1` and restarting it after `t2`).

We now present the basic rules Sieve applies to compute test plans that exercise one of the three patterns in Section 5.2.1. We later describe how Sieve avoids ineffective test plans in Section 5.2.4. Optionally, one can customize patterns by implementing new rules or manually writing test plans.

Intermediate-state rule. For a controller, Sieve generates test plans that force all possible intermediate states and exposes them to the controller. To do so, Sieve analyzes the reference controller trace and marks the sequence of state updates made by the controller within each reconciliation loop. Concretely, for every reconciliation that issues multiple state updates, $U_1, U_2, ..., U_n$, Sieve generates one test plan per state update U_i, where Sieve crashes the controller after it issues U_i. When the controller restarts after the crash, it is presented with the target intermediate state.

Stale-state rule. For stale states, Sieve generates test plans that make the controller travel

```
testWorkload: resizePVC
faults:
- faultType: crashController
  triggers:
  - triggerName: trigger1
    triggerAt: afterControllerIssues
    stateChange:
      beforeChange: 'VolCur:10GB'
      afterChange: 'VolCur:15GB'
  compositeTrigger: trigger1
```

Figure 5.5: **A test plan generated by Sieve.** This is a simplified view of the test-plan file that detected the bug in Figure 5.2. This test plan crashes the RabbitMQ controller right after the controller updates `VolCur` from 10GB to 15GB. Sieve learns every state change issued by the controller via the state-centric interfaces (e.g., `Update` in Table 5.1).

back in time and see stale states that it has already observed. Concretely, Sieve checks the controller trace for a notification-update pair (N, U), such that observing N results in an update U (see Section 5.2.4). It then searches for a subsequent state-change notification N' which has a conflicting effect with U (e.g., U deletes an object and N' creates the same object). With time traveling, if the controller mistakenly issues U after seeing the stale state N, it could corrupt the newer cluster state as notified by N'.

Sieve generates test plans that (1) block a reserved API server to prevent it from advancing its own state after it sees N, (2) after the controller sees N', time-travel the controller to see N by reconnecting the controller to the reserved stale API server, and (3) unblock the stale API server; so, the introduced staleness is only transient—both the API server and the controller catch up eventually.

Unobserved-state rule. For unobserved states, Sieve generates plans that skip states that a controller might observe during normal executions, but could potentially miss in the presence of faults. Sieve checks the controller trace to find pairs of notifications (N, N') in which N' is the closest subsequent notification that cancels the effect of N. Sieve generates test plans that (1) block the controller to prevent it from seeing N, and (2) unblock the controller when N' arrives. Such a test plan causes the controller to miss cluster states from N (inclusive) up to N' (exclusive).

5.2.4 Avoiding Ineffective Test Plans

Sieve may potentially generate a large number of test plans using rules specified in Section 5.2.3. For example, in stale-state testing, Sieve might identify every notification the controller receives as a point to inject staleness, therefore generating test plans for every received notification. For example, the naïve rule above for stale states would generate 140,000+ test plans for the MongoDB controller in Figure 5.3. It is therefore key to prune ineffective test plans.

As a guiding principle, we prune a test plan if the test plan does not introduce an intermediate-, a stale- or an unobserved-state that can affect the controller's outputs, or the introduced state is identical to states introduced by other test plans. This naturally requires Sieve to have a clear notion of what input events affect the controller's outputs.

Pruning by Causality

If a controller makes an update U based on a notification N, we consider N and U to be causally related. We consider a pair (N, U) that is not causally-related to be irrelevant from a testing standpoint, because perturbing N will not affect U.

Inferring causality between events is generally a challenge in distributed systems. By focusing on the "narrow waist" of state-centric input and output events of the controller under test, we are able to design simple yet effective rules for Sieve to infer whether a pair (N, U) is causally related. These rules are lenient and only introduce false positives at best, but not false negatives. False positives increase testing times by generating redundant test plans, whereas false negatives risk reducing test coverage that could miss bugs. While causal tracing support [140] for Kubernetes is currently in its infancy [141], we might be able to leverage it in the future.

Sieve currently considers a pair (N, U) to be causally related if both the following conditions holds (Figure 5.6 exemplifies the two causality rules):

- *Read-before-update rule:* the object pertaining to N is read by the controller before it issues U;
- *Earliest-reconciliation rule:* N and U happen in the same or adjacent reconciliation cycles. The rationale is that controllers always issue updates relevant to N in the earliest possible reconciliation cycle after N is received.

For stale- and unobserved-state testing, Sieve only generates test plans involving a notification N if it has at least one causally-related update U. We find pruning test plans

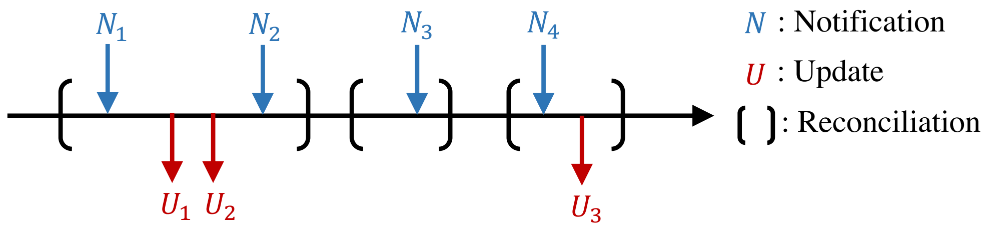


Figure 5.6: **Causality rules used by Sieve.** For simplicity, in this figure the object pertaining to each notification is immediately read by the controller. (N_1, U_1), (N_1, U_2), (N_3, U_3) and (N_4, U_3) are causally related according to the rules in Section 5.2.4. N_2 is *not* causally related to any update, given the earliest-reconciliation rule.

by causality effective, especially when there are many notifications due to other activities irrelevant to the controllers under test.

Pruning Unsuccessful Updates

Sieve ignores any update U that does not change the current cluster state. Sieve checks whether an update U is successful based on whether U triggers a state change ΔS of the cluster state. This information is typically encoded in the return value of the U operation.

For stale-state testing, Sieve further ignores an update U that, if issued again, does not change any of the subsequent cluster states, (i.e., there does not exist an N' that is affected by U). Sieve checks whether the state objects updated by U would later be conflicted with a notification N'.

The rationale for the above pruning is straightforward. If an update does not change the current cluster state, it is unlikely to cause new states in the execution. If an update U cannot affect any future cluster states, it would not perturb the controller's execution under the time-travel pattern either, i.e., if U is issued again, it would not corrupt any future state.

In practice, we found that many controllers issue unsuccessful updates that do not actually change the cluster state, including pathologically frequent ones caused by inefficient but benign behavior (see Section 5.4.2).

5.2.5 Test Plan Execution

Every test plan is executed by the Sieve test coordinator running in the testing cluster. The coordinator faithfully executes the test plan by running the test workload and injecting the faults specified in the test plan. Specifically, the test coordinator monitors state transitions of both the controller's view of the cluster state as well as the cluster state as seen by API servers. This is done based on the interposition described in Section 5.2.2; it allows the test coordinator to intercept and take actions (e.g., injecting faults) when state transitions happen. If the observed state transition matches the triggering condition ΔS specified in the test plan, the coordinator marks the condition as matched. The coordinator injects a fault (e.g., a controller crash) once all the corresponding conditions are matched. Most of the interposition and injection are done through the client APIs. But, for stale-state testing, the coordinator also needs to interpose at the API server (to make an API server stale).

As a concrete example, to execute the test plan in Figure 5.5, the test coordinator monitors every state transition issued by the RabbitMQ controller. The coordinator marks `trigger1` in the test plan as matched when it observes a state transition that updates `VolCur` from 10GB to 15GB. Since `trigger1` is the only required condition in the test plan, the coordinator injects a controller crash right after `trigger1` is matched. If the test plan specifies multiple faults, the coordinator injects them one by one according to the specified order.

The test coordinator also records the cluster states in a trace during testing, which will be compared with the reference cluster-state trace (Section 5.2.2) to detect buggy behavior.

5.2.6 Differential Test Oracles

Sieve has generic, effective oracles to automatically detect safety and liveness issues. The oracles detect buggy controller behavior based on the *cluster states* during and at the end of the test run. The goal is to validate that the testing traces are free of safety and liveness issues, in addition to monitoring anomalous controller behavior (e.g., crashes and hangs). Developers can also add domain-specific oracles.

In our experience, many buggy controller behaviors do not show immediate or obvious symptoms (e.g., crashes, hangs, and error messages). Instead, they lead to data loss, security issues, resource leaks, and unexpected application behavior which is hard to check with oracles typically used by prior art [48, 56, 142, 143, 144]; in our evaluation, only five (out of 46) bugs can be flagged by checking for exceptions or crashes.

We therefore develop differential test oracles that compare cluster states in a reference run versus those in test runs—with inconsistencies typically indicating buggy behavior. This methodology means we need to exclude nondeterministic states and state objects affected by the perturbation (Section 5.2.7).

We found that Sieve's differential oracles vastly outperform developer-written assertions in the test suites of the controllers we evaluated, because Sieve's oracles systematically examine all the state objects and their evolution during testing. It is challenging for developers to manually codify oracles that comprehensively consider the large number of relevant states.

Note that Sieve does implement regular error checks for obvious anomalies, including exceptions, error codes and timeouts. Sieve scans the controller's log and checks whether the controller encountered any unexpected exception (i.e., `panic` in Go). Sieve also checks whether the operations in the test workload return error codes or fail to complete on time.

Checking End States

Sieve systematically checks the end state after running a workload. Specifically, our oracles check the count of state objects by type and the field values of all the objects after accounting for nondeterminism (Section 5.2.7). It compares the end state of the test run versus the reference run. Sieve fails the test if it finds inconsistencies between the end states and prints human-readable messages to pinpoint inconsistencies.

Such checking is effective compared to the simpler assertions that we found in test suites for the controller projects we studied. For example, in an intermediate-state bug [145], the MongoDB controller fails to create an SSL certificate used for securing communications inside the MongoDB cluster. This causes the controller to fall back to insecure communications. Such security issues do not manifest in the form of crashes or error messages. Sieve however automatically catches the bug, because the certificate object in the faulty run does not exist in the cluster state, which is different from a normal run. The bug was detected by Sieve and confirmed by the developers.

We found that none of the 71 test cases shipped with the controller has an assertion that checks the certificate object, despite the fact that enabling TLS is recommended and is the default configuration [146]. We would not be able to repurpose the assertions in these test cases to catch this bug.

Checking State-Update Summaries

Besides the end state, Sieve also checks how the controller updates the cluster state over time. It does so by comparing summaries of constructive and destructive state updates for each object (e.g., `CREATE` and `DELETE` operations). Such checks are complementary to the end-state checks, because a correct end state does not imply that the controller behavior is always correct during the test. We find that buggy behavior can end in correct states (same as in the reference runs).

For example, in a stale-state bug [147], the XtraDB controller mistakenly deletes the front-end proxy (which routes user requests to the XtraDB cluster), causing service unavailability. After the staleness ends, the API server and the controller eventually catch up with updated states and recreate the proxy. In this case, the end state of the proxy in the test run is the same as in the reference, but the update that deleted the proxy in the test run is buggy. Sieve detects the bug by noting that the proxy pod receives 2 `CREATE` and 1 `DELETE` operations in the faulty run, but only 1 `CREATE` in the reference run.

In another intermediate-state bug [148], the NiFi controller fails to reload configuration files. The end state is the same as a normal run; however, in the faulty run, the controller did not restart the NiFi pod to reload the configuration. Sieve flags this by noting the NiFi pod receives a `CREATE` and a `DELETE` operation (to reload the configuration) in the normal run, but neither appears in the faulty run.

Note, comparing the *sequence* of state-update is unreliable and would lead to false alarms—the sequences are not strictly the same due to concurrent controller operations. The summaries instead are robust to different event orderings.

5.2.7 Dealing with Nondeterminism

The shape of a state object (the set of fields and their values) might be nondeterministic. This nondeterminism affects Sieve's test plan generation and the differential test oracles. We now describe how Sieve combats this problem.

All objects have identifying metadata (e.g., a type, namespace, and name). This is key for Sieve to identify two instances of the same object, both within a run (e.g., checking for conflicting operations in the stale-state rule) and across runs (e.g., comparing configurations of objects across runs).

Sieve identifies nondeterministic field values by running the test workloads without perturbation multiple times when generating reference runs, and then comparing the values of each field in each state object.

Objects whose identifying metadata is nondeterministic are excluded from test plan generation and subsequent steps, because Sieve cannot reliably match them across runs or setup triggering conditions for them. If other kinds of fields have nondeterministic values (typically IP addresses, timestamps, or even random port numbers), Sieve does not exclude the object but simply masks the field values. Note that these two rules still allow Sieve to spot unexpected changes to the set of fields on the object (e.g., missing deletion timestamp fields).

In addition, Sieve provides an API for Sieve users to exclude specific state objects or fields from test plans or oracles based on domain knowledge, if needed.

We implement Sieve for Kubernetes controllers. Sieve uses `kind` [149] to run a Kubernetes cluster on a single machine, so every test plan can be run entirely on one machine. Sieve configures two API servers for stale-state testing. Sieve is implemented in 5,500 lines of Python code (for test plan generation and oracles) and 3,100 lines of Go code (for automated instrumentation and fault injection).

Sieve instruments 10 API methods, representing the state-centric interface, for monitoring and perturbing states (Table 5.1). Those methods are in Kubernetes client libraries [150, 151] and the API server. Sieve implements an automated procedure to instrument the 10 methods using `dst` [152] to work with different versions of Kubernetes client libraries and API servers. Sieve analyzes the syntax tree for each method to insert monitoring and fault-injection code. Sieve applies the instrumentation when building the controller image. Sieve does not need to analyze or instrument the controller code.

API	**Component**	**Instrumentation**
`reconcileHandler`	Client	Log entry and exit
`Get`, `List`	Client	Send objects to coordinator
`Create`, `Update`, `Patch`	Client	Send objects to coordinator
`Delete`, `DeleteAllOf`	Client	Send objects to coordinator
`HandleDeltas`	Client	Send objects to coordinator
`processEvent`	API server	Send objects to coordinator
`Get`, `List`	Client	Add delay
`processEvent`	API server	Add delay

Table 5.1: **Instrumentation performed by Sieve to monitor and perturb states.** The instrumentation is automated.

In Kubernetes, level-triggered controllers do not immediately read notifications when they arrive [137]. Instead, the controller first updates a locally-cached view of the state objects; the controller reads from this cache when it uses `Get` or `List` APIs to query the cluster state. In causality analysis (Section 5.2.4) Sieve needs to know whether a notification is read before an update. To do so, Sieve analyzes the state objects updated by each notification and those read by each `Get`/`List`.

Some controllers are multi-threaded, where each thread calls a different reconcile function. Sieve uses the instrumented client libraries to obtain stacktraces whenever the controller reads or updates the cluster state (e.g., `Get`, `Create`). These stacktraces are used to differentiate between controller threads when generating and executing test plans.

Sieve's premise is that automatic and effective reliability testing for unmodified controllers is viable, by (a) exhaustively perturbing a controller's view of the cluster states and (b) using differential oracles to flag safety and liveness issues.

We validate this hypothesis with three evaluation questions: (1) Can Sieve find new bugs in real-world controllers? (2) Does Sieve do so efficiently? (3) Are Sieve's testing results trustworthy? We answer these questions in the affirmative:

- Section 5.4.1: Sieve finds *new* bugs in *all* ten evaluated controllers, resulting in a total of 46 *new* bugs, which represent a swathe of safety and liveness issues. So far, 35 of them have been confirmed and 22 have been fixed by the developers.
- Section 5.4.2: All controllers can be tested in seven hours on a cluster of 11 machines, representing a typical nightly test. This is attributed to the effective reduction techniques which reduce test plans by 46.7%–99.6% across the controllers.
- Section 5.4.3: Sieve poses a low false positive rate of 3.5%.

Tested controllers. We evaluated Sieve on ten popular controllers from the Kubernetes ecosystem for managing widely-used cloud systems (Table 5.2). The controllers are either developed by the official development team of the corresponding system, or by companies that have production-grade offerings around said systems. The term *operator* [153] in the project names refers to the Kubernetes design pattern of using a custom controller to manage an application.

Operator	**Systems**	**Developers**	**#Stars**	**#Commits**	**#WL**
cass-operator	Cassandra	DataStax	287	477	2
cassandra-operator	Cassandra	Instaclustr	227	337	2
casskop	Cassandra	Orange	177	1643	3
elastic-operator	Elasticsearch	Official	1832	3375	2
mongodb-operator	MongoDB	Percona	142	1407	5
nifikop	NiFi	Orange	101	232	3
rabbitmq-operator	RabbitMQ	Official	343	1679	3
xtradb-operator	XtraDB	Percona	302	1693	5
yugabyte-operator	YugabyteDB	Official	41	36	4
zookeeper-operator	ZooKeeper	Pravega	242	220	2

Table 5.2: **Kubernetes controllers used in our evaluation.** "#WL" stands for the number of different test workloads.

Sieve employs 2–5 basic test workloads for each controller (Table 5.2). Each workload exercises a feature of the controller. Every evaluated controller supports software deployment and autoscaling, and therefore has at least two workloads. Sieve also employs workloads for controllers that support more features, such as sharding, storage resizing, reconfiguration, and load balancing. A test workload is typically implemented in 6–12 lines of code and takes 4–12 minutes to run.

It took us on average three hours to apply Sieve to each controller, which was mostly spent on understanding how to build the controller. We expect controller developers to expend much less effort to integrate Sieve in their workflow.

5.4.1 Finding New Bugs

Controller	**Intermediate State**	**Stale State**	**Unobserved State**	**Indirect**	**Total**
cass-operator	2	1	0	0	3
cassandra-operator	0	2	1	2	5
casskop	1	2	1	0	4
elastic-operator	0	2	0	0	2
mongodb-operator	2	3	1	3	9
nifikop	2	0	0	1	3
rabbitmq-operator	1	2	1	0	4
xtradb-operator	3	3	1	0	7
yugabyte-operator	0	2	1	2	5
zookeeper-operator	0	2	1	1	4
Total	11	19	7	9	46

Table 5.3: **New bugs detected by Sieve in each controller.**

Sieve finds a total of 46 *new* bugs in the evaluated controllers (Table 5.3). Those bugs include 11 intermediate-state bugs, 19 stale-state bugs, 7 unobserved-state bugs, and 9 bugs indirectly detected by Sieve during testing. Sieve finds new bugs in *all* the evaluated controllers. We have reported all these bugs. So far, 35 of them have been confirmed and 22 have been fixed. No bug report was rejected.

Sieve can consistently reproduce all the 37 intermediate-, stale-, and unobserved-state bugs—running the test plan always reproduces the buggy behavior. In our experience, Sieve's reproducibility is invaluable for debugging test failures. It helps developers localize bugs in the source code and continuously iterate on bug fixes.

Table 5.4 shows the consequences of the 46 controller bugs and an exemplar bug for each kind of consequence. We see that many bugs have severe consequences, such as application outages, security issues, service failures, and data loss. Note that these controllers are all mature projects (Table 5.2), suggesting that controller reliability is challenging to achieve.

The bugs that Sieve finds are deep and highly unlikely to be detected by manual testing or imprecise techniques like chaos testing or randomized fault injection tools [23, 24, 25, 26]. For example, a bug [154] in nifikop is triggered only when the controller crashes between issuing two specific state updates within one reconciliation loop (the time window between the two updates is about 0.7 milliseconds). In contrast, the test workload used for detecting the bug takes about 440 seconds to finish, and causes 481 reconciliation loops and 1,687 state updates issued by the controller. Sieve can detect and consistently reproduce the bug because it relies on injecting a fault precisely when a specific cluster-state change happens.

Consequence	**Example**	**# Bugs**
Application outage	rabbitmq-operator-648: The RabbitMQ cluster is mistakenly turned down [155].	12
Service failure	K8SPSMDB-433: Sharding service for the MongoDB cluster wrongly terminated [156].	5
Data loss	K8SSAND-559: Storage volumes of Cassandra replicas wrongly deleted [157].	8
Reduced reliability	zookeeper-operator-314: The ZooKeeper cluster scaled down unexpectedly [158].	7
Misconfiguration	nifikop-49: The NiFi pod is not updated with new configuration [148].	6
Security issue	K8SPXC-896: TLS is not enabled for the XtraDB cluster [159].	6
Resource leak	cassandra-operator-398: Volumes used by deleted replicas never recycled [138].	7
Controller malfunc.	casskop-370: The controller stops serving scaling requests [94].	8

Table 5.4: **Consequences of the bugs found by Sieve (Table 5.3).** One bug can lead to multiple consequences.

Intermediate-state bugs. Sieve found 11 intermediate-state bugs. Sieve stresses a common pattern among controllers, where they issue multiple updates per reconciliation, after the controller checks for a certain condition to hold in the cluster state. However, Sieve finds bugs when these condition checks only detect states from running the reconciliation loop in its entirety; that is, when the checks do not account for intermediate states that may arise due to controller crashes. For example, in the intermediate-state bug in Figure 5.2, rabbitmq-operator compares `VolCur` and `VolDesired` to check whether the volume has been expanded already. However, this check assumes that all subsequent steps in the reconciliation succeed whenever this condition is satisfied. In the bug in Figure 2.3, the condition check cannot differentiate an intermediate state versus an unexpected faulty state. Part of the challenge is that controllers lack mechanisms analogous to write-ahead logging or journaling to guarantee atomicity of each reconcile action to enforce crash consistency. Controllers typically run as Kubernetes pods themselves and the newly created controller pod instance lacks any memory of its past execution (as they *should* – controllers must only depend on the *current state*). Sieve exposes those bugs without the need to understand source code—it systematically tests a controller with all possible intermediate states.

Stale-state bugs. Sieve found 19 stale-state bugs. In our experience, it is notoriously challenging to anticipate all possible stale states. That said, we found controllers were not adequately using Kubernetes' mechanisms to tolerate asynchrony and staleness: like object versioning and unique IDs (instead of referring to objects by names, that need not be unique), or using coordination mechanisms to enforce ordering between events. Controllers also have the option to avoid staleness by using quorum reads to API servers, but this creates a scalability bottleneck as it drives more load to etcd – developers therefore choose to synchronize selectively. In general, we do not believe there is a shortcut to reasoning about any given update under all possible staleness or time-travel scenarios. Sieve therefore aids developers by systematically testing controllers under all possible time-traveling scenarios.

Unobserved-state bugs. Sieve found 7 unobserved-state bugs. We find that all of them are rooted in latent edge-triggering behavior in the controllers, that go against the Kubernetes philosophy of designing controllers to be level-triggered (Section 5.2.1). That is, these bugs arise when the controller's correctness relies on observing a specific state transition (edges), as exemplified by Figure 5.4. By identifying states that would be later overwritten and preventing those states from being observed by the controller, Sieve is effective at exposing unobserved-state bugs in controllers.

Bugs indirectly detected by Sieve. Sieve also finds 9 bugs that were not directly triggered by input states Sieve generated but *were still correctly flagged* by its differential oracles. All these bugs could (and do) happen in reference runs as well; but because Sieve executes many test plans, some test traces inevitably *differ* from the reference traces due to these bugs, allowing Sieve to detect them. These bugs are caused by a range of issues, including (1) controllers making incorrect assumptions about the Kubernetes API (e.g., assuming a list of pods from a query have stable ordinals); (2) spurious, dangling object creations, masked by Kubernetes' garbage collection (e.g., accidentally creating ZooKeeper pods after deleting the high-level ZooKeeper cluster object); (3) the applications managed by the controller being buggy and failing. Sieve can be extended with new perturbation patterns to systematically force out some of those bugs. After understanding the root causes, we were able to reproduce two of these bugs consistently with manually written test plans.

Oracle Effectiveness

Sieve's differential oracles are crucial to detect buggy executions. Of the 46 newly found bugs, 45 were flagged by the differential test oracles. Checking logs for errors only flagged 5 bugs of which 4 were also found by our differential oracles.

Our end-state checker (Section 5.2.6) finds 28 bugs by comparing the end states of a test workload with and without perturbation. The state-update summaries checker (Section 5.2.6) finds 17 more bugs by checking the number of object updates through an execution. These oracles allow Sieve to detect bugs such as security and reliability issues (see Section 5.2.6) that do not manifest as simple failure symptoms (e.g., exceptions or crashes).

The only bug that the differential oracles fail to find but is found by a regular error check in log files, is a null-pointer dereference bug [160] that causes an unexpected controller crash. Since Kubernetes automatically restarts the controller, it does not affect the end states or the state updates.

5.4.2 Test Efficiency

Controller	**Testing Time (Machine Hours)**			**# Test Plans**
	Generation	**Execution**	**Total**	
cass-operator	0.60	43.67	44.27	218
cassandra-operator	0.49	10.72	11.21	81
casskop	0.57	12.40	12.97	125
elastic-operator	0.43	30.10	30.53	245
mongodb-operator	1.00	66.24	67.24	584
nifikop	1.17	41.61	42.78	239
rabbitmq-operator	0.47	10.60	11.07	133
xtradb-operator	1.40	62.96	64.36	395
yugabyte-operator	0.67	17.38	18.05	196
zookeeper-operator	0.33	13.75	14.08	164

Table 5.5: **Sieve's total testing time for each controller.**

Table 5.5 shows the total time Sieve takes to test each controller. All experiments were run on 11 Amazon EC2 virtual machines, each with 8-core Intel(R) Xeon(R) Platinum 8259CL CPU with 2.50GHz and 32 GB memory, running Ubuntu 20.04.2 LTS.

Sieve's total testing time varies from 11.07 to 67.24 machine hours across the controllers. Sieve runs tests in parallel because every test plan is independent. With eleven virtual machines, the total testing time for each controller is no more than 7 hours. Therefore, it is practical to run Sieve as a regular nightly test. Sieve's techniques to avoid ineffective test plans are key for tractability. Overall, Sieve prunes away 46.7%–99.6% possible test plans across the evaluated controllers.

Over 95% of the testing time is spent on executing test plans. With the perturbations introduced by Sieve, a workload takes 8.8% longer to run on average. The overhead mainly comes from delays injected by Sieve for stale- and unobserved-state testing. In a few cases, when Sieve triggers bugs that lead to liveness issues, the controller hangs and triggers a timeout (by default, 10 minutes).

Sieve also spends 0.33–1.40 hours to (1) collect the reference trace and (2) generate test plans for each controller. The collected trace for each workload contains 3,386 events of notifications, updates, or reads on average. Generating test plans takes only 20 seconds for each workload on average.

Test reduction. Sieve's techniques to avoid ineffective test plans are key for tractability. Figure 5.7 breaks down the cumulative contribution of each technique. The baseline represents the basic rules described in Section 5.2.1 without any of the pruning techniques in Section 5.2.4. Overall, Sieve prunes away 46.7%–99.6% possible test plans across the evaluated controllers.

Specifically, pruning by causality (Section 5.2.4) reduces test plans by up to 95.0% across the controllers. This reduction is particularly effective for controllers that receive many notifications that are not causally related to any update. For example, mongodb-operator receives 700+ notifications regarding 20+ state objects, which are not causally related to most of its updates. This allows Sieve to prune *136,000+* causally *unrelated* pairs of notifications and updates.

Pruning unsuccessful updates (Section 5.2.4) further prunes up to 75.8% of test plans across the controllers. In casskop, 60.0+% of updates issued by the controller do not affect the cluster state because the controller redundantly recreates two `service` objects that already exist. As none of these updates are relevant, Sieve excludes them when generating test plans.

Sieve finally prunes up to 72.9% of test plans across all controllers by focusing on deterministic triggering conditions (Section 5.2.7). This makes Sieve robust to many peculiar behaviors. For example, zookeeper-operator has an inefficient but benign behavior – it regularly clears the `NodePort` field of a `service` object in every reconciliation, forcing Kubernetes to randomly allocate a port. This leads to thousands of state transitions with random port numbers. Sieve identifies these nondeterministic transitions and avoids related test plans.

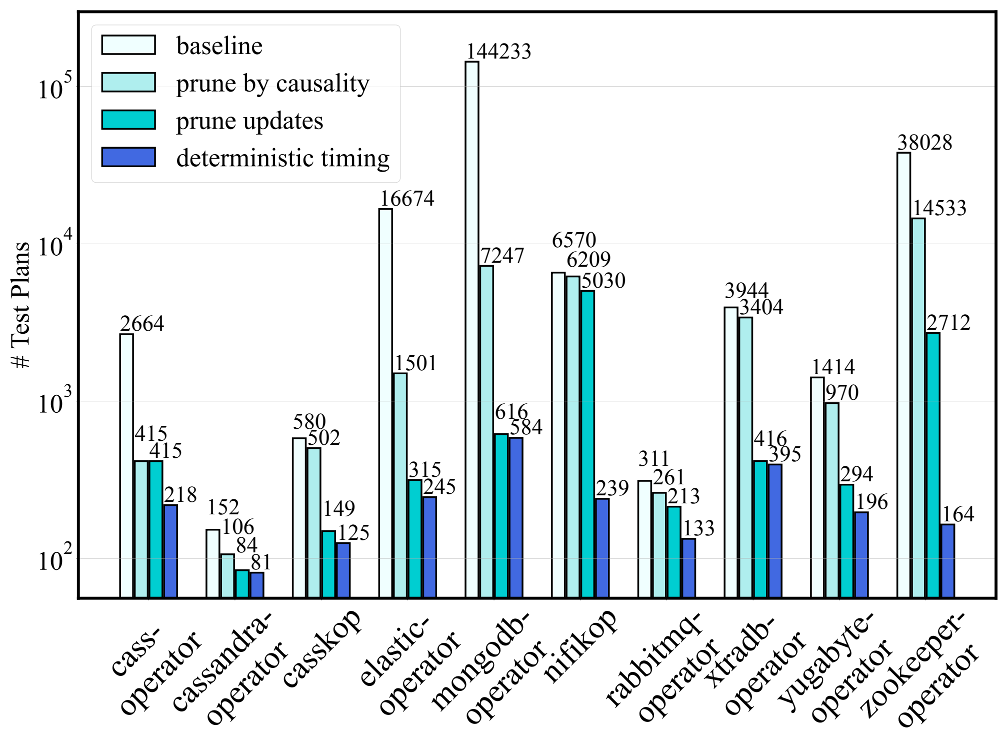


Figure 5.7: **Effectiveness of Sieve's test plan reduction techniques (Section 5.2.4).** The number of generated test plans is reduced by 46.7%–99.6% compared with the baseline.

5.4.3 False Positives

Sieve has a low false positive rate of 3.5%. It reports a total of 227 test failures for the ten evaluated controllers. 219 of them were true alarms—the test failures are caused by the 46 bugs described in Section 5.4.1 (one bug might fail multiple tests). The other eight test failures are false alarms.

The eight false alarms come from test results of three controllers (casskop, nifikop and xtradb-operator). All of them are caused by benign state transitions introduced in the faulty runs that did not happen in the reference runs.

The false alarms do not lead to opaque test failures—Sieve pinpoints the inconsistent fields. In all eight cases, the false alarms are easy to identify based on the identified fields and we could validate them by running the vanilla workload.

5.5 LIMITATIONS

Like other testing tools, Sieve is neither sound nor complete. Sieve uses specific perturbation patterns and exhaustively drives controllers to input states according to those patterns.

Sieve's differential oracles can yield both false negatives and positives. Sieve only applies its oracles on cluster states exposed by the state-centric interface. It is possible that certain application-specific states cannot be observed by the interface, which would lead to false negatives. In addition, Sieve reports false positives if the inconsistencies captured by the differential oracles are caused by benign state transitions that did not happen in the reference runs (Section 5.4.3). We found the false positive rate low (3.5%) in our evaluation.

The way Sieve deals with nondeterminism also leads to false negatives. Sieve excludes objects with nondeterministic metadata and masks nondeterministic field values in test plan generation and the differential test oracles (Section 5.2.7). This approach effectively avoids many irreproducible test plans and false positives, but also misses bugs that are triggered by states involving nondeterministic fields.

Lastly, Sieve depends on test workloads provided by the user for coverage. Implementing a test workload only takes 6-12 lines of code from our experience, but it requires domain knowledge about the controller and the system.

5.6 DISCUSSION

In this section, we reflect on our experience building Sieve and studying the root causes of bugs it found (Section 5.4.1).

We find that all the evaluated controllers adopt mature software testing practices and have numerous unit, integration, and end-to-end test cases. Some even test scenarios involving faults. However, it is prohibitively difficult for developers to anticipate all possible cluster states that may occur, let alone codify them into test cases. Sieve fills this gap by exhaustively testing input states according to patterns of interest. For two bugs, Sieve detects that the initial bug fixes are deficient in covering all the conditions. *We run Sieve on the patched controllers and Sieve still detects the bugs!*

We also find that it is challenging for developers to comprehensively check test results, given the enormous state objects and their fields. Developers typically check a few fields of interest but such assertions can easily miss subtle, but serious issues (e.g., security vulnerabilities as discussed in Section 5.2.6).

We also observe that certain bugs are likely rooted in misunderstandings of Kubernetes' design and API semantics. For example, some unobserved-state bugs are caused by incorrectly assuming that every state change can be observed by the controller; some stale-state bugs can be prevented by using Kubernetes' mechanisms like resource versioning and precondition checking. We expect such problems to be more prevalent as engineers implement more and more custom controllers for their cluster management needs.

While cloud infrastructure systems may avoid some classes of bugs, they come with hard tradeoffs. For example, not caching state objects at the controllers and API servers (Figure 5.1) could avoid stale-state bugs. However, it would introduce significant performance overheads to the controllers (memory accesses become network round trips) and make the data store a scalability bottleneck [133]. Also, transactions are not a solution for intermediate-state bugs – it would complicate the state-centric interface and prevent controllers from independently making progress regardless of failures, a key factor for resilience.

Since there is no silver bullet to implementing reliable controllers, we believe that automatic tools like Sieve are critical to cluster management reliability.

5.7 SUMMARY

This chapter presents Sieve an automatic reliability testing tool for controllers. The key idea of Sieve is state perturbation: perturbing a controller's view of the cluster state in ways the controller is expected to tolerant, and comparing the cluster state's evolution with and without perturbations to automatically detect safety and liveness issues. We find that Sieve is effective and practical by implementing state perturbation with three different perturbation patterns. Sieve's usability and reproducibility play a critical role in understanding, debugging, and fixing reliability bugs. Sieve's testing technique is general and easy to extend – it separates the policy (how to perturb a controller's view of state) from mechanisms (how to realize perturbations). Hence, we are able to use the technique to detect a wide range of bugs without brittle heuristics, specifications or hypotheses. Our goal is to make Sieve a part-and-parcel of every controller developers' toolkit, and to harden the growing number of controllers that power today's data centers. We have made Sieve publicly available at `https://github.com/sieve-project/sieve`.

CHAPTER 6: RELATED WORK

In this chapter, we discuss previous work related to this dissertation. Section 6.1 discusses related work on verification, and Section 6.2 discusses related work on testing.

6.1 VERIFICATION

Verification is an old idea, and there is a long line of research on systems verification. In this section, we discuss two types of verification closely related to this dissertation: deductive verification and model checking.

Deductive verification is an approach that proves system correctness with mathematical proofs that connect systems to their formal specifications. Deductive verification often requires manual proof effort and involves proof assistants or verifiers for checking proofs. Anvil belongs to the class of deductive verification. Anvil emphasizes verifying both liveness and safety properties for practical controller implementations. In addition, ESR is the first formal specification for controller correctness.

Model checking guarantees system correctness by exhaustively exploring a system's state space. Model checking is automatic and does not require manual proof effort, but often suffers from the state-explosion problem. Sieve is not a model checking tool, but it bears similarities to model checking, in that Sieve drives a system to a range of states to find bugs. Different from previous work, Sieve is designed for state-reconciliation system implementations and it explores implementation state space at the level of the cluster state's evolution. Sieve trades exhaustiveness for efficiency.

6.1.1 Deductive Verification

There has been a lot of progress in using deductive verification to build correct systems, including operating systems [30, 31], compilers [32], file and storage systems [33, 34, 35, 36, 37, 38, 39, 40], and distributed systems [41, 42, 43]. Despite the rich literature, most systems verification efforts so far focus on safety rather than liveness [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 42, 43, 44, 161, 162, 163]. A notable exception is IronFleet [41], which also verifies liveness of system implementations.

IronFleet presents a methodology for building distributed systems and verifying their liveness and safety properties by combining Hoare-style and TLA-style reasoning. Anvil is inspired by IronFleet's methodology, but 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 6.1), 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 4.13).

```
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 6.1: **A representative liveness lemma example from IronFleet (written in Dafny) [164].** 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.

Recent work has proposed techniques for automating liveness verification. Ivy [125, 165] incorporates a technique for proving liveness of distributed protocols using first-order logic [166, 167]. 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 [168] 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 [169, 170, 171], 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 [172] and liveness of concurrent programs [173, 174, 175]. 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.

6.1.2 Model Checking

Model checking has been applied for verifying critical system software [53, 57, 58, 59, 60, 61, 62, 176, 177, 178, 179, 180, 181, 182], including operating systems, file systems, and distributed systems. Due to the state-explosion problem, model checking for system implementations is often performed with a timeout or within a bounded space—the model checker exhaustively explores a finite subset of the entire state space by limiting the number of events.

Compared to model checking, Anvil requires manual proof effort, but achieves stronger correctness guarantee by verifying that *all* possible controller behaviors are correct. In particular, Kivi [182] is closely related to Anvil. Kivi focuses on detecting failures caused by desired state changes, configuration errors, and controller interactions by using SPIN to model check if controllers in a specific deployment violate user-supplied properties at the model level (not implementation level). Kivi performs bounded model checking by limiting depth and time. In contrast, Anvil allows developers to verify controller *implementations* against ESR, a *general* controller-correctness specification, with manual proof effort.

Sieve is not a verification tool but it bears similarities to implementation-level model checking [53, 57, 58, 59, 60, 61, 62], in that we drive an unmodified implementation to a range of states to find bugs. Unlike model checking, Sieve does not seek to exhaustively cover the controller's state space. It instead executes developer-supplied test cases and exhaustively perturbs these test cases according to some fault patterns. Additionally, model checkers typically rely on a specification for correct behavior. While Sieve intentionally does not require hand-crafted specifications, it leans on reference traces as a partial specification of expected correct behavior.

6.2 TESTING

Previous research has made great progress in testing distributed systems [45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62], but they do not offer a generally applicable, comprehensive, and efficient testing approach for diverse controllers.

Fault-injection tools [45, 46, 47, 48, 49, 50, 51, 52, 183] have been developed for distributed systems, including chaos testing tools from the industry for cloud infrastructure systems [23, 24, 25, 26]. Sieve's goals differ from those in the fault injection literature. Sieve seeks to expose controllers to as many input states as possible to test their reliability. For us, faults just happen to be a good mechanism to drive controllers to the required states. Compared to randomized chaos testing approaches that are unaware of cluster state transitions, Sieve can precisely force specific bug-triggering state transitions and consistently reproduce bugs. Furthermore, unlike prior art [48, 49, 184], Sieve is not based on an expert's hypotheses about vulnerable regions in the code under test.

A few prior tools can, in principle, expose some bugs found by Sieve. For example, concurrency-testing tools [53, 54, 55, 56] may expose bugs triggered by unobserved states (which in essence occur due to reordering of events). Similarly, tools that check for crash safety [48, 49, 185, 186] could expose bugs caused by the intermediate states. Finally, tools that inject network partitions [50], with expert guidance, could find some bugs caused by the stale-state pattern, i.e., a partition might force a controller to talk to a lagging API server (after talking to an up-to-date one). In contrast to these tools, Sieve does not target one class of bugs. Through exhaustive state perturbations, Sieve finds many kinds of bugs, essentially combining the power of prior targeted tools. Further, the chances that prior tools will find the bugs Sieve does are small, as they lack the context required to efficiently drive controllers to their buggy corners (e.g., a network-partition injector is unlikely to reliably orchestrate time-travel bugs).

Acto [65], which is also our work but not part of this dissertation, is closely related to Sieve. Acto is designed for testing operators—a special type of controllers for managing applications deployed on cloud infrastructure systems. Acto focuses on generating and mutating desired state descriptions and checks three operation correctness requirements: the operator (1) always reconciles the managed application to desired states, (2) performs managed application recovery from undesired or error states by rolling back to a previous good state, and (3) should be resilient to misoperations (i.e., operation errors) by preventing them from driving the application into error states. Acto and Sieve are complementary to each other: Acto can automatically generate testing workloads for Sieve by generating a series of desired state descriptions, and Sieve can enhance Acto's tests using state perturbation that injects external events (e.g., crashes). We have applied the combination of Acto and Sieve to test the verified controllers of Anvil, and the testing discovered bugs in unverified code.

CHAPTER 7: CONCLUSION AND FUTURE WORK

Reliability of cloud infrastructure systems is critical but challenging. This dissertation focuses on verification and testing techniques for controllers—the core components of cloud infrastructure systems. The key idea of this dissertation is to develop formal verification and comprehensive, efficient testing techniques that are generally applicable to diverse controllers using state-centric reasoning.

We first present state-centric reasoning, an approach for reasoning about behaviors of diverse controllers without knowing their implementation details. State-centric reasoning provides a uniform representation of controllers' behaviors by leveraging an important opportunity in cloud infrastructure systems: There is a clean separation between controllers and the cluster state, and the cluster state is represented as highly introspectable objects. This idea of reasoning about the cluster state's evolution enables this dissertation's work on specifying, verifying, and testing controllers.

To enable formal verification for cloud infrastructure systems, we present eventually stable reconciliation (ESR), the first general formal specification for controllers. The key idea of ESR is to capture controllers' essential functionality using liveness. ESR is formalized concisely in Temporal Logic of Actions (TLA). Our analysis shows that ESR precludes 69% of all the bugs detected by Sieve and Acto, two state-of-the-art controller testing tools developed by us.

We then present Anvil, the first framework for implementing practical, formally verified controllers. Anvil employs a hybrid of Hoare-style and TLA-style reasoning to allow developers to prove liveness and safety properties of controller implementations. With Anvil, we have built three Kubernetes controllers and verified that they implement ESR. The verified controllers are practical—they achieve feature parity and competitive performance compared to existing, unverified references. ESR and Anvil enable a practical approach that gradually verifies cloud infrastructure systems by incrementally replacing existing controllers with verified ones.

Formal verification provides strong correctness guarantees, but the approach of building new, provably correct systems does not directly improve the reliability of existing systems. To improve the reliability of existing controllers before verified replacements are available, we then present Sieve, an automatic reliability testing tool for controllers. The key idea of Sieve is state perturbation, a general testing approach that perturbs a controller's view of the cluster state and uses differential oracles to catch bugs. Sieve employs three different perturbation patterns and exhaustively introduces state perturbations. Sieve is effective in finding new bugs—it has detected 46 new bugs in 10 popular Kubernetes controllers.

Many open challenges remain in improving the reliability of cloud infrastructure systems. We discuss future work that continues along the lines of this dissertation.

How to reduce proof effort for verifying liveness of system implementations? Anvil provides verification support but the verification process still involves a lot of manual effort (proof-to-code ratio ranges from 4.5 to 7.4). Most of the existing techniques for proof automation focus on safety instead of liveness [122, 123, 124, 127, 129]. Some techniques, such as LVR [168] and dynamic abstraction [166, 167], have successfully automated liveness verification for distributed protocols such as Paxos. An interesting future direction is to develop proof automation techniques for TLA-style liveness verification for system implementations.

How to verify liveness properties (e.g., ESR) for systems that are not written as state machines? To apply TLA-style verification, we have to structure systems implementations as state machines. However, most of the systems are not written as state machines explicitly. It is an interesting challenge to close the gap between such system implementations and liveness verification. A potential approach is to extend recent techniques [36, 39, 40, 42] for verifying concurrent and distributed systems implementations with liveness verification support and apply them to verify existing Kubernetes controller implementations. Another approach is to faithfully transpile between procedural code and state machine models like PlusCal [187] and PGo [188].

How to combine the strengths of testing and verification? Lightweight formal methods have been increasingly used in industry to improve production system reliability [189, 190, 191, 192]. Lightweight formal methods are much easier to apply compared to proof-based formal verification and provide stronger guarantees compared to regular testing. Lightweight formal methods include a broad range of techniques such as property-based testing and (bounded) model checking. A starting point is to extend Sieve to be a bounded model checking tool that exhaustively searches both event interleaving and input space.

REFERENCES

[1] “Kubernetes,” https://kubernetes.io/, 2023.

[2] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-Scale Cluster Management at Google with Borg,” in *Proceedings of the 10th European Conference on Computer Systems (EuroSys’15)*, Apr. 2015.

[3] C. Tang, K. Yu, K. Veeraraghavan, J. Kaldor, S. Michelson, T. Kooburat, A. Anbudurai, M. Clark, K. Gogia, L. Cheng, B. Christensen, A. Gartrell, M. Khutornenko, S. Kulkarni, M. Pawlowski, T. Pelkonen, A. Rodrigues, R. Tibrewal, V. Venkatesan, and P. Zhang, “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.

[4] “Latest Kubernetes Adoption Statistics: Global Insights and Analysis for 2025,” https://edgedelta.com/company/blog/kubernetes-adoption-statistics.

[5] Kubernetes-59848, “Kubernetes is vulnerable to stale reads, violating critical pod safety guarantees,” https://github.com/kubernetes/kubernetes/issues/59848, Feb. 2018.

[6] “Kubernetes stale reads,” https://kubernetespodcast.com/episode/218-k8s-stale-reads/.

[7] “Comment on Kubernetes-59848: Kubernetes is vulnerable to stale reads, violating critical pod safety guarantees,” https://github.com/kubernetes/kubernetes/issues/59848#issuecomment-525833106, Aug. 2019.

[8] Kubernetes-56261, “Scheduler should delete a node from its cache if it gets ”node not found” error,” https://github.com/kubernetes/kubernetes/issues/56261, Nov. 2017.

[9] “Need to handle stale kubernetes watch cache,” https://github.com/argoproj/argo-cd/issues/1605, 2019.

[10] Cassandra-operator-400, “Cassandra node can be decommissioned wrongly which blocks scale down,” https://github.com/instaclustr/cassandra-operator/issues/400, Jan. 2021.

[11] Cassandra-operator-402, “PVC can be accidentally deleted when controller reads stale data from apiserver,” https://github.com/instaclustr/cassandra-operator/issues/402, Jan. 2021.

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

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

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

[15] H. Kumar and J. Šafránek, "Storage on Kubernetes - Learning From Failures," https://youtu.be/zuGjw595OiQ?t=853, Nov. 2019.

[16] G. Templeton and S. Davidson, "How a Couple of Characters (and GitOps) Brought Down Our Site," https://youtu.be/FiEm2zOuHsg?t=922, May 2022.

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

[18] M. Cebula and B. Sherrod, "10 Weird Ways to Blow Up Your Kubernetes," https://youtu.be/FrQ8Lwm9_j8?t=1192, Nov. 2019.

[19] "Kubernetes persistent volume controller," https://github.com/kubernetes/kubernetes/blob/master/pkg/controller/volume/persistentvolume/pv_controller.go.

[20] "Fix pv reclaim failed due to its phase is wrongly updated to the failed state by kcm," https://github.com/kubernetes/kubernetes/pull/125767.

[21] "If a pvc has an empty storageclass name, don't try to assign a default storageclass," https://github.com/kubernetes/kubernetes/pull/122704.

[22] "storage class assignment should not ignore errors," https://github.com/kubernetes/kubernetes/pull/117064.

[23] "Chaoskube: chaoskube periodically kills random pods in your kubernetes cluster," https://github.com/linki/chaoskube, 2020.

[24] "Kubemonkey: An implementation of netflix's chaos monkey for kubernetes clusters," https://github.com/asobti/kube-monkey, 2020.

[25] "Pumba: Chaos testing, network emulation, and stress testing tool for containers," https://github.com/alexei-led/pumba, 2020.

[26] "Chaos mesh — a solution for system resiliency on kubernetes," https://dzone.com/articles/chaos-mesh-a-chaos-engineering-solution-for-system, 2020.

[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[28] E. Goldweber, W. Yu, S. A. V. Ghahani, and M. Kapritsos, "IronSpec: Increasing the Reliability of Formal Specifications," in *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (OSDI'24)*, July 2024.

[29] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, “An Empirical Study on the Correctness of Formally Verified Distributed Systems,” in *Proceedings of the 12th European Conference on Computer Systems (EuroSys’17)*, Apr. 2017.

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “SeL4: Formal Verification of an OS Kernel,” in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP’09)*, Oct. 2009.

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

[32] X. Leroy, “Formal Verification of a Realistic Compiler,” *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, July 2009.

[33] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, “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.

[34] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, “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.

[35] H. Chen, T. Chajed, A. Konradi, S. Wang, A. Ileri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, “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.

[36] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, “Verifying concurrent, crash-safe systems with Perennial,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP’19)*, Oct. 2019.

[37] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen, “Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP’19)*, Oct. 2019.

[38] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno, “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.

[39] T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich, “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.

[40] T. Chajed, J. Tassarotti, M. Theng, M. F. Kaashoek, and N. Zeldovich, "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.

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

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

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

[44] Y.-S. Chang, R. Jung, U. Sharma, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "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.

[45] A. Tseitlin, "The Antifragile Organization," *Communications of the ACM*, vol. 56, no. 8, pp. 40–44, Aug. 2013.

[46] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos Engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, Mar. 2016.

[47] R. Majumdar and F. Niksic, "Why is Random Testing Effective for Partition Tolerance Bugs?" in *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)*, Jan. 2018.

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

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

[50] "Jepsen," https://jepsen.io/, 2020.

[51] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Feb. 2017.

[52] C. Drăgoi, C. Enea, B. K. Ozkan, R. Majumdar, and F. Niksic, "Testing Consensus Implementations Using Communication Closure," in *Proceedings of 2020 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'20)*, Nov. 2020.

[53] M. Musuvathi, S. Qadeer, and T. Ball, "CHESS: A Systematic Testing Tool for Concurrent Software," Tech. Rep. MSR-TR-2007-149, November 2007. [Online]. Available: https://www.microsoft.com/en-us/research/publication/chess-a-systematic-testing-tool-for-concurrent-software/

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

[55] B. K. Ozkan, R. Majumdar, and S. Oraee, "Trace Aware Random Testing for Distributed Systems," in *Proceedings of 2019 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'19)*, Oct. 2019.

[56] X. Yuan and J. Yang, "Effective Concurrency Testing for Distributed Systems," in *Proceedings of the 25th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'20)*, Mar. 2020.

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

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

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

[60] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "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.

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

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

[63] L. Lamport, “The Temporal Logic of Actions,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, May 1994.

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

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

[66] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM 12*, 1969.

[67] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the Presence of Partial Synchrony,” *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[68] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel, “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.

[69] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzel, J. Howell, J. R. Lorch, O. Padon, and B. Parno, “Verus: A Practical Foundation for Systems Verification,” in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP’24)*, Nov. 2024.

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

[71] “Kubernetes Components,” https://kubernetes.io/docs/concepts/overview/components/, 2021.

[72] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: Flexible, Scalable Schedulers for Large Compute Clusters,” in *Proceedings of the 8th European Conference on Computer Systems (EuroSys’13)*, Apr. 2013.

[73] "vSphere: Unified Management for Containers and VMs," https://www.vmware.com/products/vsphere.html, 2021.

[74] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, May 2016.

[75] C. Hall, "AWS, Google, Microsoft, Red Hat's New Registry to Act as Clearing House for Kubernetes Operators," https://www.datacenterknowledge.com/open-source/aws-google-microsoft-red-hats-new-registry-act-clearing-house-kubernetes-operators, Mar. 2019.

[76] J. Pipes, M. Hausenblas, and N. Taber, "Introducing the AWS Controllers for Kubernetes (ACK)," https://aws.amazon.com/cn/blogs/containers/aws-controllers-for-kubernetes-ack/, Aug. 2020.

[77] C. Sosa and P. Bhatia, "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.

[78] Z. Tang, X. Li, and F. Guo, "Demystifying Kubernetes as a service - How Alibaba cloud manages 10,000s of Kubernetes clusters," https://www.cncf.io/blog/2019/12/12/demystifying-kubernetes-as-a-service-how-does-alibaba-cloud-manage-10000s-of-kubernetes-clusters/, Dec. 2019.

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

[80] C. MacCárthaigh, "PID loops and the art of keeping systems stable," https://www.infoq.com/presentations/pid-loops/, June 2019.

[81] S. Haase, "How an Operator Becomes the Hero of the Edge," in *OperatorCon*, May 2019.

[82] M. Musaji, "Why Operators are essential for Kubernetes," https://www.redhat.com/en/blog/why-operators-are-essential-kubernetes, Apr. 2021.

[83] P. Ratis, "Lessons Learned using the Operator Pattern to build a Kubernetes Platform," in *USENIX SREcon*, Oct. 2021.

[84] "etcd," https://etcd.io/.

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

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

[87] "OpenShift: The developer and operations friendly Kubernetes distro," https://github.com/openshift, 2021.

[88] "Introducing the AWS Controllers for Kubernetes (ACK)," https://aws.amazon.com/blogs/containers/aws-controllers-for-kubernetes-ack.

[89] "AWS Controllers for Kubernetes (ACK)," https://github.com/aws-controllers-k8s/community.

[90] "The Istio service mesh," https://istio.io.

[91] "Crossplane: The cloud native control plane framework," https://www.crossplane.io.

[92] Y. Mao, Z. Chen, Y. Zhang, M. Wang, Y. Fang, G. Zhang, R. Shi, and R. T. B. Ma, "StreamOps: Cloud-Native Runtime Management for Streaming Services in ByteDance," *Proc. VLDB Endow.*, vol. 16, no. 12, 2023.

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

[94] casskop-370, "[BUG] Casskop fails to clean up PVCs and refuses to handle user requests after crash and restart," https://github.com/Orange-OpenSource/casskop/issues/370, 2021.

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

[96] A. Pnueli, "The Temporal Logic of Programs," in *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, Oct. 1977.

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

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

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

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

[101] "Apache zookeeper," https://zookeeper.apache.org/.

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

[103] L. De Moura and N. Bjørner, "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.

[104] K. R. M. Leino, "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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[121] "FluentBit," https://fluentbit.io/, 2023.

[122] J. Yao, R. Tao, R. Gu, and J. Nieh, "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.

[123] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan, "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.

[124] T. Hance, M. Heule, R. Martins, and B. Parno, "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.

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

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

[127] H. Ma, A. Goel, J.-B. Jeannin, M. Kapritsos, B. Kasikci, and K. A. Sakallah, "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.

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

[129] A. Goel and K. Sakallah, "On Symmetry and Quantification: A New Approach to Verify Distributed Protocols," in *Proceedings of the 13th NASA Formal Methods Symposium (NFM'21)*, May 2021.

[130] C. Lou, P. Huang, and S. Smith, "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.

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

[132] rabbitmq-operator-782, "[BUG] PVC expansion fails if the controller crashes in the middle of a reconciliation," https://github.com/rabbitmq/cluster-operator/issues/782, 2021.

[133] M. Brooker, “The Fundamental Mechanism of Scaling,” http://brooker.co.za/blog/2021/01/22/cloud-scale.html, 2020.

[134] “Resource versions,” https://kubernetes.io/docs/reference/using-api/api-concepts/#resource-versions, 2021.

[135] K8SPSMDB-430, “[BUG] Stale deletion timestamps lead to undesired statefulset and PVC deletion,” https://jira.percona.com/browse/K8SPSMDB-430, 2021.

[136] A. Alpar, “Using Finalizers to Control Deletion,” https://kubernetes.io/blog/2021/05/14/using-finalizers-to-control-deletion/, May 2021.

[137] “What is a Level Based API,” https://book-v1.book.kubebuilder.io/basics/what_is_a_controller.html, 2023.

[138] cassandra-operator-398, “[BUG] Reconcilation fails to delete PVCs if missing a deletion timestamp of the Cassandra pod,” https://github.com/instaclustr/cassandra-operator/issues/398, 2021.

[139] “How finalizers work,” https://kubernetes.io/docs/concepts/overview/working-with-objects/finalizers/#how-finalizers-work, 2021.

[140] J. Mace, R. Roelke, and R. Fonseca, “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems,” in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP’15)*, Oct. 2015.

[141] “API Server Tracing,” https://github.com/kubernetes/kubernetes/pull/94942, 2021.

[142] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan, “Understanding and Detecting Software Upgrade Failures in Distributed Systems,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP ’21)*, Oct. 2021.

[143] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, “Early Detection of Configuration Errors to Reduce Failure Damage,” in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI’16)*, Nov. 2016.

[144] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, “Testing Configuration Changes in Context to Prevent Production Failures,” in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI’20)*, Nov. 2020.

[145] K8SPSMDB-578, “[BUG] Failure of creating SSL-internal certificates when the controller crashes and restarts at some particular point,” https://jira.percona.com/browse/K8SPSMDB-578, 2021.

[146] Percona Distribution for MongoDB Operator Documentation, “Transport Layer Security (TLS),” https://www.percona.com/doc/kubernetes-operator-for-psmongodb/TLS.html, 2021.

[147] K8SPXC-725, "[BUG] HAproxy stateful set and services get mistakenly deleted when reading stale spec.haproxy.enabled," https://jira.percona.com/browse/K8SPXC-725, 2021.

[148] nifikop-49, "[BUG] NiFi configuration cannot be reloaded if the controller crashes and restarts in the middle of a reconciliation," https://github.com/konpyutaika/nifikop/issues/49, 2021.

[149] "kind: Kubernetes IN Docker - local clusters for testing Kubernetes," https://kind.sigs.k8s.io/, 2021.

[150] "kubernetes-sigs/controller-runtime," https://github.com/kubernetes-sigs/controller-runtime, 2021.

[151] "kubernetes/client-go," https://github.com/kubernetes/client-go, 2021.

[152] "Decorated Syntax Tree," https://github.com/dave/dst, 2021.

[153] "Kubernetes Operators," https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, 2021.

[154] nifikop-79, "[BUG] Nifikop fails to scale down NiFi cluster due to a crash in the middle of reconcileNifiPod," https://github.com/konpyutaika/nifikop/issues/79, 2022.

[155] rabbitmq-operator-648, "[BUG] Reading stale RabbitMQ cluster information leads to unexpected StatefulSet deletion," https://github.com/rabbitmq/cluster-operator/issues/648, 2021.

[156] K8SPSMDB-433, "[BUG] Sharding stateful set gets mistakenly deleted when reading stale field values," https://jira.percona.com/browse/K8SPSMDB-433, 2021.

[157] K8SSAND-559, "[BUG] PVCs can be deleted mistakenly when reading stale deletion timestamp information," https://k8ssandra.atlassian.net/browse/K8SSAND-559, 2021.

[158] zookeeper-operator-314, "[BUG] Reading stale ZooKeeper cluster status can lead to undesired pod and PVC deletion," https://github.com/pravega/zookeeper-operator/issues/314, 2021.

[159] K8SPXC-896, "[BUG] The controller fails to set up SSL-internal certificates if a crash happens at some particular point," https://jira.percona.com/browse/K8SPXC-896, 2021.

[160] K8SPSMDB-434, "[BUG] Nil pointer dereference when reconfiguring spec.sharding.enabled," https://jira.percona.com/browse/K8SPSMDB-434, 2021.

[161] J. R. Lorch, Y. Chen, M. Kapritsos, B. Parno, S. Qadeer, U. Sharma, J. R. Wilcox, and X. Zhao, "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.

[162] T. Hance, Y. Zhou, A. Lattuada, R. Achermann, A. Conway, R. Stutsman, G. Zellweger, C. Hawblitzel, J. Howell, and B. Parno, "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.

[163] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos, "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.

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

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

[166] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, "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.

[167] O. Padon, J. Hoenicke, K. L. McMillan, A. Podelski, M. Sagiv, and S. Shoham, "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.

[168] J. Yao, R. Tao, R. Gu, and J. Nieh, "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.

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

[170] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo, "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)*, Sep. 2018.

[171] N. Macedo, J. Brunel, D. Chemouil, and A. Cunha, "Pardinus: A temporal relational model finder," *J. Autom. Reason.*, vol. 66, no. 4, pp. 861–904, 2022. [Online]. Available: https://doi.org/10.1007/s10817-022-09642-2

[172] B. Cook, A. Podelski, and A. Rybalchenko, "Proving Program Termination," *Communications of the ACM*, vol. 54, no. 5, pp. 88–98, May 2011.

[173] A. Farzan, Z. Kincaid, and A. Podelski, "Proving Liveness of Parameterized Programs," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'16)*, July 2016.

[174] P. Baumann, R. Majumdar, R. S. Thinniyam, and G. Zetzsche, "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.

[175] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis, "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.

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

[177] L. Ouyang, X. Sun, R. Tang, Y. Huang, M. Jivrajani, X. Ma, and T. Xu, "Multi-Grained Specifications for Distributed System Model Checking and Verification," in *Proceedings of the 20th European Conference on Computer Systems (EuroSys'25)*, Mar. 2025.

[178] X. S. L. H. Y. H. Ruize Tang, Minghua Wang and X. Ma, "Converos: Practical Model Checking for Verifying Rust OS Kernel Concurrency," in *Proceedings of the 2025 USENIX Conference on USENIX Annual Technical Conference (ATC'25)*, July 2025.

[179] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Dec. 2002.

[180] J. Yang, C. Sar, and D. Engler, "EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, Nov. 2006.

[181] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using Model Checking to Find Serious File System Errors," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.

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

[183] Y. Chen, X. Sun, S. Nath, Z. Yang, and T. Xu, ""push-button reliability testing for cloud-backed applications with rainmaker"," in *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, Apr. 2023.

[184] D. Yuan, Y. Luo, X. Zhuang, G. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Oct. 2014.

[185] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Oct. 2014.

[186] R. Alagappan, A. Ganesan, Y. Patel, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Correlated Crash Vulnerabilities," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Nov. 2016.

[187] L. Lamport, "The PlusCal Algorithm Language," in *International Colloquium on Theoretical Aspects of Computing (ICTAC'09)*, July 2009.

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

[189] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services Uses Formal Methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015.

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

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

[192] M. A. Kuppe, "Validating System Executions with the TLA+ Tools," in *TLA+ Conference*, Apr. 2024, https://youtu.be/NZmON-XmrkI.

APPENDIX A: WRITING TLA-STYLE LIVENESS PROOF USING ANVIL

To support liveness verification, Anvil provides a TLA embedding and a set of TLA proof rule lemmas that can be used for temporal logic reasoning for any system. We use a simple example to demonstrate how to use Anvil's TLA embedding and lemmas to prove liveness. We consider a program with two threads competing to acquire the same lock and then release the lock. We model the program as a state machine in Verus.

We model the program state using a `lock` variable to represent whether the lock is acquired and a map `threads` to represent the states of the two threads.

```
struct ProgramState {
    lock: bool,
    threads: Map<Tid, PC>,
}
```

Each thread (`A` or `B`) can be (1) `Waiting` for the lock, (2) `Holding` the lock, or (3) `Terminated` after it releases the lock.

```
enum Tid {
    A,
    B
}

enum PC {
    Waiting,
    Holding,
    Terminated,
}
```

In the initial state, the lock is not acquired and both threads are in the `Waiting` state.

```
spec fn init() -> StatePred<ProgramState> {
    |s: ProgramState| {
        &&& !s.lock
        &&& s.threads.contains_key(Tid::A)
        &&& s.threads[Tid::A] is Waiting
        &&& s.threads.contains_key(Tid::B)
        &&& s.threads[Tid::B] is Waiting
    }
}
```

Note that `init` returns a state predicate, i.e., a closure that takes a `ProgramState` and returns a `bool`.

The transition function is defined as a number of possible choices for the next step of the program. First, a thread might acquire the lock in the next step. A next step is defined as an `Action` with a `precondition` and a `transition`. The step takes only when the `precondition` is satisfied and it drives the program to a new state returned by `transition`.

```
spec fn thread_acquires_lock() -> Action<ProgramState, Tid, ()> {
    Action {
        precondition: |tid: Tid, s: ProgramState| {
            !s.lock && s.threads[tid] is Waiting
        },
        transition: |tid: Tid, s: ProgramState| {
            (ProgramState {
                lock: true,
                threads: s.threads.insert(tid, PC::Holding)
            }, ())
        },
    }
}
```

This step takes a `Tid` as input, checks whether the lock is ready to acquire (`lock` is false) and the thread is in `Waiting` state, and then in the next state changes `lock` to true and the thread state to `Holding`.

Second, a thread might release the lock in the next step.

```
spec fn thread_releases_lock() -> Action<ProgramState, Tid, ()> {
    Action {
        precondition: |tid: Tid, s: ProgramState| {
            s.threads[tid] is Holding
        },
        transition: |tid: Tid, s: ProgramState| {
            (ProgramState {
                lock: false,
                threads: s.threads.insert(tid, PC::Terminated)
            }, ())
        },
    }
}
```

This step takes a `Tid` as input, checks whether the thread is in `Holding` state, and then in the next state changes `lock` to false and the thread state to `Terminated`.

Third, the state machine takes a stutter step and the state remains unchanged.

```
spec fn stutter() -> Action<ProgramState, (), ()> {
    Action {
        precondition: |input: (), s: ProgramState| { true },
        transition: |input: (), s: ProgramState| { (s, ()) },
    }
}
```

It is a convention in TLA to add a stutter step to a state machine so that the state machine can run forever (by repeating the stutter step) since an execution is defined as an *infinite* sequence of states.

The overall transition function for the state machine is defined in `next`. In each step, the state machine might choose to 1) let thread `A` or `B` acquire the lock (if the precondition holds), 2) let thread `A` or `B` release the lock (if the precondition holds), or 3) do nothing.

```
spec fn next() -> ActionPred<ProgramState> {
    |s, s_prime: ProgramState| {
        ||| thread_acquires_lock().forward(Tid::A)(s, s_prime)
        ||| thread_acquires_lock().forward(Tid::B)(s, s_prime)
        ||| thread_releases_lock().forward(Tid::A)(s, s_prime)
        ||| thread_releases_lock().forward(Tid::B)(s, s_prime)
        ||| stutter().forward(())(s, s_prime)
    }
}
```

Our goal is to prove a liveness property: Both threads eventually terminate.

```
spec fn both_terminated() -> StatePred<ProgramState> {
    |s: ProgramState| s.threads[Tid::A] is Terminated && s.
       threads[Tid::B] is Terminated
}
```

We create a proof obligation: Assuming that a state machine model starts with the initial state and continues running the next step, and the two steps for acquiring and releasing locks are weakly fair, how to prove that both threads eventually terminate? Note that the weak fairness assumptions state that for both thread `A` and `B`, if the action to let the thread acquire or release the lock remains enabled, then the action eventually occurs.

```
proof fn liveness_proof(model: TempPred<ProgramState>)
```

```
    requires
        model.entails(lift(init())),
        model.entails(always(lift(next()))),
        model.entails(tla_forall(|tid| thread_acquires_lock().
           weak_fairness(tid))),
        model.entails(tla_forall(|tid| thread_releases_lock().
           weak_fairness(tid))),
    ensures
        model.entails(eventually(lift(both_terminated())))
{ ... }
```

The key to prove this liveness property is to reason about the progress of the two threads by applying the `wf1` lemma. We first prove that starting from the initial state, eventually one of the threads is holding the lock.

```
let one_holding = |s: ProgramState| {
    ||| s.threads[Tid::A] is Holding && s.threads[Tid::B] is
       Waiting && s.lock
    ||| s.threads[Tid::A] is Waiting && s.threads[Tid::B] is
       Holding && s.lock
};
use_tla_forall(model, |tid| thread_acquires_lock().weak_fairness(
    tid), Tid::A);
// model ⊨ thread_acquires_lock().weak_fairness(Tid::A)
thread_acquires_lock().wf1(Tid::A, model, next(), init(),
    one_holding);
// model ⊨ init ⇝ one_holding
```

The `wf1` rule states that "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 an action 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. In this case, P is the initial state, and Q is the state where one of the threads is holding the lock. We choose A to be the action that thread `A` acquires the lock. The intuition of the proof behind `wf1` is that, if the action A happens upon the initial state, then thread `A` is holding the lock in the next state. If other actions happen upon the initial state, then either no thread is holding the lock in the next state (if the stutter step happens), or thread `B` is holding the lock in the next state (if the action that thread `B` acquires the lock happens); the actions for releasing the lock cannot happen because their preconditions are not satisfied by the initial state.

To apply `wf1`, we only need to instantiate the quantified weak fairness condition on thread id `A` to prove precondition (4). The other preconditions are automatically proved by Verus.

After proving that the initial state leads to a state where one of the threads is holding the lock, we need to split the case on *which thread is holding the lock*. We start with the first case where thread `A` is holding the lock. The execution after this state becomes deterministic: `A` needs to first release the lock (and terminates), then `B` can acquire the lock and then release it (and terminates). We perform a series of `wf1` application and use transitivity of the leads-to operator to prove that from the state where thread `A` is holding the lock, eventually both threads terminate.

```
let ta_holding_tb_waiting = |s: ProgramState| s.threads[Tid::A]
    is Holding && s.threads[Tid::B] is Waiting && s.lock;
let ta_terminated_tb_waiting = |s: ProgramState| s.threads[Tid::A
    ] is Terminated && s.threads[Tid::B] is Waiting && !s.lock;
let ta_terminated_tb_holding = |s: ProgramState| s.threads[Tid::A
    ] is Terminated && s.threads[Tid::B] is Holding && s.lock;
...
thread_releases_lock().wf1(Tid::A, model, next(),
    ta_holding_tb_waiting, ta_terminated_tb_waiting);
// model ⊨ ta_holding_tb_waiting ⇝ ta_terminated_tb_waiting
...
thread_acquires_lock().wf1(Tid::B, model, next(),
    ta_terminated_tb_waiting, ta_terminated_tb_holding);
// model ⊨ ta_terminated_tb_waiting ⇝ ta_terminated_tb_holding
...
thread_releases_lock().wf1(Tid::B, model, next(),
    ta_terminated_tb_holding, both_terminated());
// model ⊨ ta_terminated_tb_holding ⇝ both_terminated
leads_to_transitive_n!(
    model,
    lift(ta_holding_tb_waiting),
    lift(ta_terminated_tb_waiting),
    lift(ta_terminated_tb_holding),
    lift(both_terminated())
);
// model ⊨ ta_holding_tb_waiting ⇝ both_terminated
```

For the second case, we prove that from the state where thread `B` is holding the lock, eventually both threads terminate. The proof is done in a similar way to the first case.

We then combine the two cases to show that from the state where thread `A` or `B` is holding the lock, eventually both threads terminate.

```
// model ⊨ ta_holding_tb_waiting ⇝ both_terminated
// model ⊨ tb_holding_ta_waiting ⇝ both_terminated
or_leads_to_combine(model, lift(ta_holding_tb_waiting), lift(
    tb_holding_ta_waiting), lift(both_terminated()));
// model ⊨ (ta_holding_tb_waiting ∨ tb_holding_ta_waiting) ⇝ both_terminated
temp_pred_equality(
     lift(ta_holding_tb_waiting).or(lift(tb_holding_ta_waiting)),
     lift(one_holding)
); // (ta_holding_tb_waiting ∨ tb_holding_ta_waiting) = one_holding
// model ⊨ one_holding ⇝ both_terminated
```

Finally, we apply leads-to transitivity again to show that the initial state leads to the state where both threads terminate.

```
leads_to_transitive(model, lift(init()), lift(one_holding), lift(
    both_terminated()));
// model ⊨ init ⇝ both_terminated
leads_to_apply(model, lift(init()), lift(both_terminated()));
// model ⊨ ◊both_terminated
```

This example demonstrates how to use Anvil's TLA embedding and lemmas to prove liveness for state machine models. Developers can also connect the proof to executable code by structuring the code as an executable state machine and proving that the implementation state machine conforms to the model using Verus' Hoare-style reasoning.