
# Inductive Invariants That Spark Joy: Using Invariant Taxonomies to Streamline Distributed Protocol Proofs

Tony Nuda Zhang
University of Michigan

Travis Hance
Carnegie Mellon University

Manos Kapritsos
University of Michigan

Tej Chajed
University of Wisconsin–Madison

Bryan Parno
Carnegie Mellon University

## Abstract

Proving the correctness of a distributed protocol is a challenging endeavor. Central to this task is finding an *inductive invariant* for the protocol. Currently, automated invariant inference algorithms require developers to describe protocols using a restricted logic. If the developer wants to prove a protocol expressed without these restrictions, they must devise an inductive invariant manually.

We propose an approach that simplifies and partially automates finding the inductive invariant of a distributed protocol, as well as proving that it really is an invariant. The key insight is to identify an *invariant taxonomy* that divides invariants into Regular Invariants, which have one of a few simple low-level structures, and Protocol Invariants, which capture the higher-level host relationships that make the protocol work.

Building on the insight of this taxonomy, we describe the Kondo methodology for proving the correctness of a distributed protocol modeled as a state machine. The developer first manually devises the Protocol Invariants by proving a *synchronous* version of the protocol correct. In this simpler version, sends and receives are replaced with atomic variable assignments. The Kondo tool then automatically generates the asynchronous protocol description, Regular Invariants, and proofs that the Regular Invariants are inductive on their own. Finally, Kondo combines these with the synchronous proof into a *draft* proof of the asynchronous protocol, which may then require a small amount of user effort to complete. Our evaluation shows that Kondo reduces developer effort for a wide variety of distributed protocols.

## 1 Introduction

Distributed protocols are notoriously difficult to reason about. Because they underpin critical applications and infrastructure, any bugs can have severe consequences. Hence, in recent years, both researchers [14,20,25,37,38,41,45] and practitioners [2,30,46] have turned to formal verification to rigorously prove the correctness of distributed systems and protocols.

At the core of a formal distributed protocol safety proof is an *inductive invariant*, which implies that a desired safety property holds throughout a system's execution. As argued in previous work [13,26,43,44], manually deriving inductive invariants is a creative challenge. For example, the IronFleet [14,15] authors reported spending months to identify and prove an inductive invariant for Paxos [22,23].

Unsurprisingly, this challenge spurred a new category of algorithms and tools to automatically find the inductive invariants of distributed protocols [10,13,17–19,26,33,43,44]. For instance, DuoAI [43] finds an inductive invariant for Paxos in minutes, without any user guidance.

However, automated invariant inference has its own Achilles' heel—it limits how developers may express their protocols. State-of-the-art tools like DuoAI only work when the problem of checking the correctness of inductive invariants is a decidable one. Thus, they apply only to protocols that operate within the confines of a first-order logic fragment known as *effectively propositional reasoning* (EPR [34]). As an example of its limited expressivity, EPR does not permit arithmetic, requiring developers to use creative abstractions to encode common systems primitives such as epoch numbers and vote counting. As detailed by Padon et al. [31], expressing a protocol such as Paxos in EPR is quite challenging.

In summary, the current state of the art is a landscape of two extremes: the developer has to choose between 1) expressing the protocol naturally using standard programming primitives such as arithmetic, but manually find the inductive invariant, or 2) abstracting the protocol into the restrictive confines of EPR so as to apply automated invariant finding tools.

In this work, we present a new approach to bridge this gap. Our key insight is that there is an *invariant taxonomy* in the clauses within an inductive invariant. This taxonomy can be used to modularize invariants and proofs into strata of varying difficulty. Furthermore, we observe that all but the most difficult stratum can be derived almost fully automatically, even in a non-EPR setting that permits intuitive programming constructs such as arithmetic. Interestingly, proving the top-most

stratum is often equivalent to proving the safety of a simpler, synchronous version of the protocol.

In this taxonomy, we identify a class of **Regular Invariants** with simple, regular structure that stem from recurring features and patterns in asynchronous message-passing systems. These invariants relate messages to their sending or receiving hosts (dubbed *Message Invariants*), assert the monotonic nature of common data structures (*Monotonicity Invariants*), and govern the ownership of unique resources (*Ownership Invariants*). As we will detail in subsequent sections, these invariants are not only easy to derive, but also easy to prove.

On the other hand, there is a separate class of **Protocol Invariants** that deal with the global relationships between hosts in the system. These capture the macro-level operation of the particular protocol, reflecting the structure of its design, and thus may require careful developer thought. Such invariants might state, for example, that a decision is made only when a majority of the nodes agree on it, or that all replicas of a log must have agreeing entries.

This taxonomy is not merely conceptual, but has significant practical implications. First, it enables a streamlined, systematic approach to finding and proving inductive invariants. The developer can easily dispatch Regular Invariants, before using them as building blocks for proving Protocol Invariants. Such an approach modularizes the derivation and proof of invariants into distinct components, with the most challenging part, Protocol Invariants, neatly contained. This is in contrast to monolithic proofs of the past, where developers have written invariants that intertwine complex protocol logic with simple local-level reasoning, thereby proliferating the difficulty of Protocol Invariants across the entire proof.

More importantly, Regular Invariants are sufficiently systematic that they can be derived from the protocol description with a few hints from the user, and then proven automatically, even in a general purpose verification tool such as Dafny [24]. Although our proposed taxonomy does not cover the space of all possible invariants, we observe that these derived Regular Invariants, in conjunction with a set of user crafted Protocol Invariants, often suffice to prove a wide variety of distributed protocols. In fact, these Protocol Invariants are typically (although not always) the same set needed to prove a *synchronous* version of the protocol; i.e., in a model without a network, where a sender sends a message and the receiver receives it in one atomic step.

Leveraging these insights, we design Kondo, a methodology and tool that lets developers harness the structure afforded by the invariant taxonomy. Using Kondo, the developer focuses their efforts on proving the correctness of a simpler, synchronous version of the protocol. Kondo then generates the asynchronous protocol from the synchronous description, and automatically devises and proves Regular Invariants for the asynchronous protocol with a few simple hints from the user. In addition, Kondo carries over Protocol Invariants from the synchronous proof, and combines them with Regular Invariants to create a *draft* proof of the asynchronous protocol. Sometimes, this draft proof is complete and constitutes the final proof; otherwise, the user helps complete the proof by adding some proof annotations to the draft.

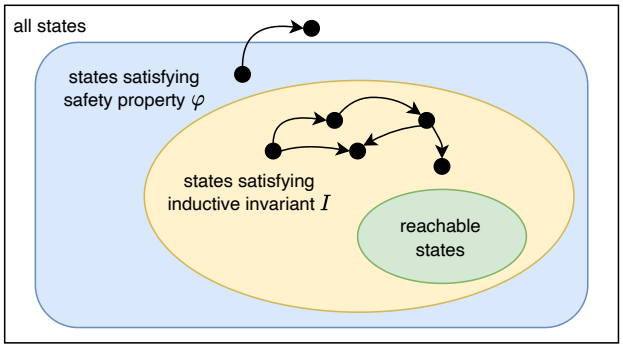


Figure 1: Venn diagram of states in a distributed system. The dots and arrows represent example states and transitions.

Overall, we make the following contributions.

1. We identify an invariant taxonomy that distinguishes between Regular Invariants and Protocol Invariants, and define three sub-classes of Regular Invariants (Section 3).
2. We propose the Kondo approach to help developers leverage the structure afforded by the invariant taxonomy, and implement the Kondo tool [1] as a new feature in the Dafny compiler. The user begins by proving a simpler, synchronous protocol. Kondo then aids in lifting that proof to a fully asynchronous setting (Sections 4 and 5).
3. We evaluate the effectiveness of Kondo on a range of distributed protocols that span different application domains, from consensus to mutual exclusion. We show that Kondo can simplify finding and proving the inductive invariants of these protocols, and identify areas in which Kondo may be less effective (Section 6).

## 2 The Challenge of Inductive Invariants

Manually proving the correctness of an asynchronous distributed protocol is widely regarded as a challenging task. Its difficulty is compounded by the lack of principled techniques for structuring the invariants that a developer must derive for the proof. This results in invariants that entangle complex protocol logic with otherwise standard network semantics. Such invariants are hard to reason about and can complicate the entire proof.

**Background.** A correctness proof of a protocol involves showing that a desired safety property φ is an *invariant* that is true in all reachable states of the system. However, φ itself is usually too weak to support an inductive argument; i.e., there exist states satisfying φ that can transition to an unsafe state (Figure 1). As a result, the developer must devise an *inductive*

```
datatype Vote = Yes | No
datatype Decision = Abort | Commit
datatype Message =
    VoteReqMsg
    | VoteMsg(v: Vote, src: nat)
    | DecideMsg(d: Decision)
datatype Option<T> = None | Some(v: T)
datatype Coordinator = Variables(
    numParticipants: nat,              // some constant N
    decision: Option<Decision>,        // initially None
    yesVotes: set<nat>,                // initially empty
    noVotes: set<nat>                  // initially empty
)
datatype Participant = Variables(
    hostId: nat,                       // unique identifier ∈ [0,N)
    preference: Vote,                  // non-deterministic constant
    decision: Option<Decision>,        // initially None
)
```

Figure 2: Hosts and message states of the Two-Phase Commit protocol, written in Dafny. Note that Vote, Decision and Message are sum types.

*invariant* $I = I_1 \wedge \cdots \wedge I_n$, composed of several individual invariants $I_k$. $I$ needs to both imply $\varphi$ and be *inductive*, i.e., it should hold in all initial states of the system, and be closed under system transitions—if $I$ holds for a state $s$, it also holds for the next state $s'$ after any transition from $s$. If an individual conjunct $I_k$ is itself inductive (even if it does not imply $\varphi$) we refer to it as *self-inductive*.

Coming up with an inductive invariant is a creative challenge because it must be strong enough to be inductive, yet weak enough to encompass all the reachable states of the system. For distributed protocols, this challenge is exacerbated by the presence of an asynchronous network that may arbitrarily delay, drop, duplicate, or re-order messages. In addition to considering the local states of each host, the developer must also contend with the state of the network and its interaction with hosts. As a result, proofs of distributed protocols require complex inductive invariants involving many clauses that simultaneously juggle host and network states [14, 28, 31].

## 2.1 Case Study: Two-Phase Commit

To highlight the challenge in finding inductive invariants, we use the classic Two-Phase Commit protocol. It is parameterized by an arbitrary number of participants, and it has a single coordinator (Figure 2). Participants are initialized with some preference of Yes or No that they communicate to the coordinator, which then makes a decision, using the protocol listed in Figure 3. The safety specification we target in this

1. Coordinator broadcasts VoteReqMsg.
2. Upon receiving VoteReqMsg, a participant $p$ replies VoteMsg($p$.preference, $p$.hostId).
3. Upon receiving VoteMsg($v$, *src*), the coordinator adds *src* to its *yesVotes* or *noVotes* set based on $v$.
4. The coordinator waits to receive votes from every participant. Then, if the coordinator has $|noVotes| > 0$, it sets its local decision to Abort and broadcasts DecideMsg(Abort). Otherwise, if $|yesVotes| = numParticipants$, it sets its decision to Commit and broadcasts DecideMsg(Commit).
5. Upon receiving DecideMsg($d$), a participant sets its local decision to $d$.

Figure 3: Two-Phase Commit protocol description.

example is that if any participant reaches a Commit decision, then every participant's local *preference* must be Yes:

$$\forall id : participants[id].\text{decision} = \text{Some}(\text{Commit}) \implies \big(\forall id' : participants[id'].\text{preference} = \text{Yes}\big) \quad \text{(2PC-Safety)}$$

Unfortunately, outside the context of EPR, there is no established methodology one can follow in finding an inductive invariant for this specification. Instead, they must rely solely on wit and will, in a journey of intuition-guided trial and error. In particular, the developer devises a candidate list of other protocol properties that when taken in conjunction with 2PC-Safety, they suspect, will form an inductive invariant.

An example of an invariant one may come up with is:

> "*if there is a* DecideMsg(Commit) *in the network, then every* VoteMsg *from each participant must contain* Yes."

Despite its apparent correctness, proving this invariant requires non-trivial supporting invariants. The developer will find that the proof requires host-level reasoning about how the coordinator processes votes, how it decides based on its local vote tally, and how it avoids sending conflicting messages. Overall, this leads to a proof that tightly intertwines host and network reasoning.

In this work, we argue that such monolithic invariants need not be the norm. Rather, they can be structured into forms that are more tractable. The idea of using simple invariants as building blocks for proving more complex ones is not a new one [3, 4]. However, we propose a systematic way of applying this idea to distributed protocols and derive invariants in layers of increasing complexity.

# 3 The Invariant Taxonomy

We present an invariant taxonomy designed to help developers tease apart host-level invariants from low-level invariants (e.g.,

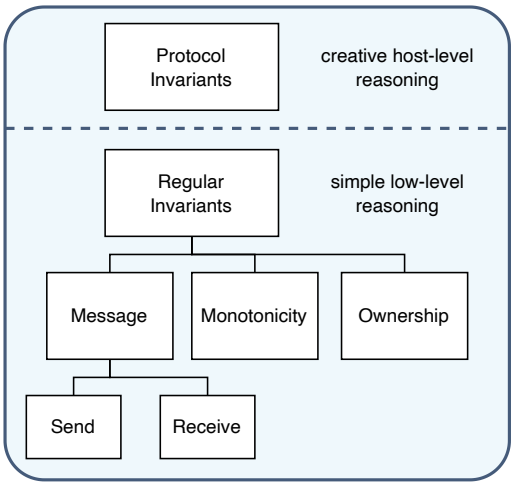


Figure 4: The invariant taxonomy.

invariants about the network). This taxonomy has two distinct categories—an upper stratum of Protocol Invariants, and a lower stratum of Regular Invariants, illustrated in Figure 4. Regular Invariants are easy to devise mechanically, and thus they can be assumed while discovering Protocol Invariants (which we show in Section 4 can even be discovered while completely ignoring the network). Using this taxonomy, the developer can contain host-level Protocol Invariants—the portion that demands user creativity—inside a well-demarcated portion of the proof. In turn, the Regular Invariants supporting those invariants are uncontaminated by host-level logic.

Note that these categories do not cover the space of all invariants. For a given protocol, there can exist invariants that are neither Regular Invariants nor Protocol Invariants. However, we find that our categorization is comprehensive enough to encompass all the invariants needed to prove a wide variety of distributed protocols (Section 6).

## 3.1 Regular Invariants

Regular Invariants are structurally simple and can be derived without requiring an understanding of why the protocol works. They concern low-level properties that follow from network semantics, the monotonicity of data like certain counters and sets, and the syntactic structure of protocol steps. They are also often self-inductive, which makes them easy to prove, and are even amenable to automation (Sections 4 and 5).

We identify three subcategories of Regular Invariants, namely Message Invariants, Monotonicity Invariants and Ownership Invariants, depicted by the hierarchy in Figure 4.

**Message Invariants.** These relate the state of the network to the state of hosts. In this way, they act as the logical bridge for proving invariants about relationships between host states when the hosts are separated by a network medium. Message Invariants come in two flavors:

1. **Send Invariants** assert that a message $m$ is in the network only if it was sent by a host. They also describe, for each message variant, some relationship $p$ between the message contents and the state of its sender:

$$\forall m \in network : p(m, hosts[m.\text{src}])$$

2. **Receive Invariants** assert that if some condition $q$ is met at some host $h$, then there must exist some message $m$ in the network that was received by that host and has some relationship $r$ with the host:

$$\forall h : q(hosts[h]) \implies \big(\exists m : m \in network \wedge h = m.\text{dst} \wedge r(hosts[h], m)\big)$$

**Monotonicity Invariants.** Monotonic data types are a common primitive in distributed protocols [38]. Widely-used structures include grow-only epoch counters to filter stale messages, and add-only sets to collect votes from participant nodes. Monotonicity Invariants capture the monotonic properties of these data types, by asserting how the state of individual hosts may evolve as the system transitions:

$$\forall \sigma, \sigma' : lteq(\sigma, \sigma')$$

Here, $\sigma$ and $\sigma'$ are respectively the prior and current states of a host, and *lteq* represents some ordering relation on the values of local variables between the states.

Monotonicity Invariants require referencing the past states of hosts, which are typically not part of the distributed system model. In Section 4.2, we describe our history-preservation technique that enables us to systematically transform a protocol into one that preserves information about state histories. The protocol augmented with history information supports stating and proving Monotonicity Invariants, and using them in the proofs of higher-level Protocol Invariants.

**Ownership Invariants.** Many distributed protocols also require reasoning about uniquely owned resources. For example, at most one client can hold a unique lock in a distributed lock system, or at most one host can hold a unique key in a sharded key-value store.

Ownership Invariants capture the semantics of such resources. Specifically, they say that for each unique resource $\gamma$, at most one node can 'own' $\gamma$ at any point in time, and if $\gamma$ is in transit in the network, then no nodes can have ownership of $\gamma$. These properties are common to protocols that deal with resource ownership.

## 3.2 Protocol Invariants

Protocol Invariants describe a relationship $\ell$ among hosts, and do not mention the network. That is, they have the form

$$\forall h_1, \ldots, h_n : \ell(hosts[h_1], \ldots, hosts[h_n])$$

As such, Protocol Invariants are ignorant of the complications arising from network asynchrony. Instead, they focus on the higher-level reasoning of how host behaviors culminate in the overall correctness of the protocol, thus capturing properties that require insight into *why* the protocol is correct.

In addition, observe that given an asynchronous distributed protocol $\mathcal{P}$, any Protocol Invariant in $\mathcal{P}$ must also be an invariant in the *synchronous* version of the protocol $\mathcal{P}_{sync}$. This version, $\mathcal{P}_{sync}$, is one in which messages are delivered instantaneously between hosts—a sender sending a message and the receiver receiving it occurs as one atomic step, without a network delay. In practice, we observe that invariants used in proving the safety property in $\mathcal{P}_{sync}$ tend to be helpful Protocol Invariants in $\mathcal{P}$. Moreover, in all the protocols we evaluated, these Protocol Invariants, in conjunction with a set of automatically derived Regular Invariants, are all that is needed to prove $\mathcal{P}$ (Section 6.2).

### 3.3 Streamlining Proofs Using the Taxonomy

The invariant taxonomy mirrors the unique roles played by the network and the protocol logic. The network, while an inevitable part of reasoning about distributed systems, does not contribute to the underlying logic of the protocol—it simply serves as a medium to carry information between hosts. Instead, it is the interplay of host actions that affects the outcome of the protocol.

Respecting this natural division is key to writing a modular and efficient proof. By confining creativity-demanding invariants to exclusively describe hosts, Protocol Invariants ensure that user creativity is called upon only when needed, while the remaining mundane Regular Invariants are dispatched with minimal effort.

**Two-Phase Commit Revisited.** We revisit the Two-Phase Commit example from Section 2.1 to demonstrate how one can use the invariant taxonomy to bring order and simplicity to their proof. Figure 5 lists a set of invariants for Two-Phase Commit. The conjunction of these invariants and 2PC-Safety forms an inductive invariant that proves 2PC-Safety.

Of the six invariants, only *A*5 and *A*6 are Protocol Invariants. Discovering them involves protocol-level insight about how the states of different hosts are related. On the other hand, the remaining invariants are Regular Invariants. They arise from the individual steps where messages are sent or received, and describe what that particular step says about the network. For instance, *A*3 stems directly from protocol step 4. Likewise, *A*4 follows from step 5. They are also self-inductive—each invariant is preserved by the step that it directly relates to and is trivially preserved by other steps. For example, *A*1 relates directly to protocol step 2, which is the only step that adds a VOTEMSG to the network.

Armed with this insight, the developer can employ the following proof strategy. They can first write down the Regular

| | |
|---|---|
| *A*1 | For each VOTEMSG($v$, *src*) in the network, *src* is a valid participant identifier. |
| *A*2 | For each VOTEMSG($v$, *src*) in the network, $v$ reflects the *preference* of the participant identified by *src*. |
| *A*3 | For each DECIDEMSG($d$) in the network, $d$ reflects the local *decision* at the coordinator. |
| *A*4 | For each participant that decided Commit, DECIDEMSG(Commit) must be a message in the network. |
| *A*5 | For each *id* in *yesVotes* at the coordinator, the participant identified by *id* must have the corresponding *preference*. |
| *A*6 | If the coordinator decided Commit, then every participant's preference must be Yes. |

Figure 5: Two-Phase Commit protocol invariants structured using the invariant taxonomy. The conjunction of these, together with 2PC-Safety, forms the inductive invariant of the protocol. *A*5 and *A*6 are Protocol Invariants, while the rest are Message Invariants.

Invariants *A*1, *A*2, *A*3 and *A*4 without any thought about overall protocol correctness. This is easy because these invariants are apparent just from looking at individual, local protocol steps. Their self-inductive nature also allows the developer to quickly check if the invariants they wrote are correct. Finally, with these Regular Invariants effortlessly in place, the developer then devises the crowning jewels *A*5 and *A*6 as Protocol Invariants, the one part of the proof that requires creativity.

## 4 Finding Invariants the Kondo Way

We now present the Kondo methodology and tool to help developers leverage the regularity afforded by the invariant taxonomy. In contrast to EPR-based approaches, Kondo targets general protocol models where determining the inductiveness of an invariant is an undecidable problem.

Kondo is based on two core observations. First, the systematic structure of Regular Invariants makes both *deriving and proving* these invariants amenable to automation, even in an undecidable setting that permits higher-order logic.

More surprisingly, we observe that Protocol Invariants can be devised and proven in a synchronous version of the protocol $\mathcal{P}_{sync}$ and used directly in the asynchronous protocol. Because messages are delivered instantaneously between hosts in $\mathcal{P}_{sync}$, it is much easier to iteratively devise an invariant and prove inductiveness. In all the distributed protocols in our evaluation (Section 6.2), the Protocol Invariants taken from $\mathcal{P}_{sync}$, in conjunction with the set of derived Regular Invariants for the asynchronous protocol, are sufficient to form an inductive invariant for each protocol's safety property (although this may not always be the case; see Section 6.5).

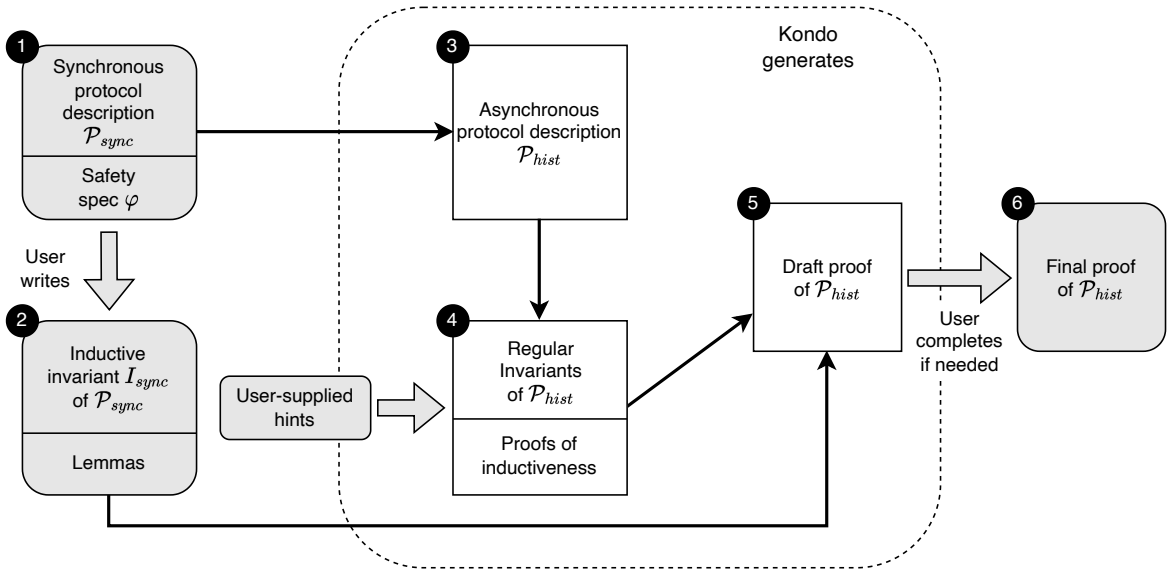


Figure 6: Kondo workflow. Shaded bubbles represent artifacts that the user writes. Boxes represent artifacts that Kondo generates. Sometimes, the draft proof from ❺ may constitute the final proof, and the user need not perform step ❻ in such cases.

These observations inform the Kondo methodology. Using Kondo, the developer first writes and proves a synchronous version of the protocol. Then, Kondo automatically generates

1. a *history-preserving* asynchronous protocol $\mathcal{P}_{hist}$. In addition to modeling an asynchronous network, $\mathcal{P}_{hist}$ maintains a history of host states that aid in expressing Message and Monotonicity Invariants;
2. a set of Regular Invariants (sometimes with the help of user-supplied hints), along with their proofs of inductiveness; and
3. a *draft* proof of correctness of $\mathcal{P}_{hist}$. This draft may constitute the final proof, or may require modest effort from the developer to complete.

## 4.1 Overview

Figure 6 shows an overview of how a developer uses the Kondo methodology and tool to prove the safety of a protocol.

Step ❶: The user begins by writing a synchronous version of the protocol, denoted as $\mathcal{P}_{sync}$, together with a safety specification φ. This synchronous execution model does not have a network component. Instead, the overall system is simply a collection of host states that communicate atomically.

Step ❷: The user proves that $\mathcal{P}_{sync}$ satisfies φ by devising an *inductive invariant* $I_{sync}$ that implies φ. Because we operate in a general setting in which checking the inductiveness of $I_{sync}$ is undecidable, the user may also need to write a set of lemmas to convince the verifier that $I_{sync}$ is indeed inductive.

Step ❸: Given $\mathcal{P}_{sync}$, the Kondo tool automatically generates a history-preserving asynchronous protocol $\mathcal{P}_{hist}$. It shares the same safety property φ as $\mathcal{P}_{sync}$.

Step ❹: Kondo generates a set of Regular Invariants for $\mathcal{P}_{hist}$, together with lemmas that prove their inductiveness. We denote the conjunction of all Regular Invariants as $R$. Note that to generate Receive, Monotonicity and Ownership Invariants, Kondo requires small hints from the user. The nature of these hints is detailed in Section 5.1.

Step ❺: From the user-written proof of $\mathcal{P}_{sync}$, Kondo generates a draft proof of $\mathcal{P}_{hist}$. This draft uses the conjunction $I_{\text{sync}} \wedge R$ as the inductive invariant. It lifts lemmas written for the $\mathcal{P}_{sync}$ proof to the new asynchronous context, while leaving gaps in places where the translation fails. Section 5.2 describes this process. Notably, code generated by Kondo is human-readable.

Step ❻: The user runs the verifier on the draft proof. In some cases, the draft suffices as the final proof. Otherwise, particular lemmas in the draft may be incomplete. The user then manually completes the proof of $\mathcal{P}_{hist}$ by filling in gaps in the bodies of these lemmas. Notably, *no new lemmas* need to be constructed, and the logical line of reasoning is identical to $\mathcal{P}_{sync}$. However, the user may need to write additional proof annotations to convince the verifier that the lemmas still hold in the asynchronous protocol.

## 4.2 Protocol Models

Like prior work [13, 14, 26, 43, 44], we use the temporal logic of actions (TLA [21]) to model a protocol as a state machine described by non-deterministic transition relations. This state machine in turn contains one or more sets of hosts. For example, the Paxos system has three sets of Proposer, Acceptor and Learner hosts respectively.

Hosts are themselves modeled as event-driven state machines that communicate via messages. A host can 1) take a spontaneous action that may or may not send a message, 2) take an action given some received message as input, or 3) crash for an indefinite amount of time.

Formally, a host is defined by a HOSTINIT($h$: Host) predicate and a HOSTNEXT($h$: Host, $h'$: Host) relation. HOSTINIT circumscribes the initial states of the host, while HOSTNEXT describes the host's state transition relation.

In the Kondo methodology, the developer works with two versions of a distributed protocol, $\mathcal{P}_{sync}$ and $\mathcal{P}_{hist}$, that differ by their network model. $\mathcal{P}_{sync}$ is initially written by the developer, whereas $\mathcal{P}_{hist}$ is derived automatically.

**Synchronous Protocol $\mathcal{P}_{sync}$.** The global state of $\mathcal{P}_{sync}$ is $S := (\sigma_1, \ldots, \sigma_n)$, an $n$-tuple of host states. Its initial states are defined by the predicate

$$\textsc{SyncInit}(S) := (\textsc{HostInit}(\sigma_1), \ldots, \textsc{HostInit}(\sigma_n))$$

asserting that hosts satisfy their respective initial conditions.

The system transitions are defined through the relation

$$\textsc{SyncNext}(S, S') := \textsc{Action}_1(S, S') \vee \cdots \vee \textsc{Action}_K(S, S')$$

where each action disjunct represents an atomic transition that the system may take. Each action also falls in one of two categories. First, one non-deterministically chosen host may take a local action, i.e., one that doesn't send or receive any data. Second, a non-deterministically chosen pair of sender and recipient hosts may take a corresponding send and receive action simultaneously; i.e., the sender transmits a message that is received instantaneously by the recipient.

Note that a consequence of tightly coupling sender and recipient pairs is that one host cannot both receive and send messages within a single action, as that would allow an arbitrary chain of hosts taking steps at once. This limitation does not sacrifice generality, as an action that receives and sends may always be modeled as two consecutive actions, with the first receiving the message and the second sending its response. It is, however, an additional restriction over the host model in prior work [14,15], and may increase proof complexity. Nevertheless, our evaluation shows that even with such a restriction, Kondo allows users to write simpler proofs than previous state of the art (see Paxos discussion in Section 6.3).

**History-Preserving Asynchronous Protocol $\mathcal{P}_{hist}$.** Given the synchronous protocol $\mathcal{P}_{sync}$, Kondo automatically generates a *history-preserving* asynchronous protocol $\mathcal{P}_{hist}$. This model adds to the synchronous model an asynchronous network that may arbitrarily delay, drop, duplicate, or re-order messages. Like prior work [14,15,45], we model this network as a monotonically increasing set of sent messages. When a host sends a message, the message is added to this set. When a host calls receive, it retrieves from this set some message addressed to it.

Unique to Kondo is the history-preserving aspect of $\mathcal{P}_{hist}$. It maintains a sequence *history* of host snapshots, enabling us to express monotonicity properties. Formally, let $history := [S_0, \ldots, S_m]$ be a sequence of host state $n$-tuples. Each entry in *history* is a snapshot of all hosts in the system. Then the global state of $\mathcal{P}_{hist}$ is $S_{hist} := (history, N)$. The latest entry in *history* represents the current state of the hosts, while $N$ is the set of messages representing the network's latest state.

The initial states of the system are defined by

$$\begin{aligned}\textsc{Init}(S_{hist}) := \; & len(S_{hist}.history) = 1 \\ & \wedge \textsc{SyncInit}(S_{hist}.history[0]) \wedge S_{hist}.N = \emptyset\end{aligned}$$

The system transitions are given by the relation

$$\begin{aligned}\textsc{Next}(S_{hist}, S'_{hist}) := \\ & \wedge len(S_{hist}.history) \geq 1 \\ & \wedge S_{hist}.history = trunc(S'_{hist}.history) \\ & \wedge \textsc{SyncNext}(curr(S_{hist}), curr(S'_{hist}))\end{aligned}$$

Here, $trunc(s)$ yields $s$ with the last item removed. Meanwhile, $curr(S_{hist})$ gives the current state of the system, namely the tuple $(last(S_{hist}.history), S_{hist}.N)$, where $last(s)$ returns the last item in a sequence $s$.

## 4.3 Case Study: Echo Server

To illustrate the Kondo methodology, we apply it to a simple Echo Server protocol. It comprises an arbitrary number of clients and a single server. Each client maintains a constant set of *requests* that are defined by unique client and request identifiers.

Clients send their requests to the server. Upon receiving a request, the server responds by echoing the request back to the sender. When a client receives a response, it stores it in a local *responses* set. The safety specification is that clients do not receive rogue responses; i.e. for each client $c$, every element in $c$.responses matches a request in $c$.requests:

$$\forall\, client: \; client.\text{responses} \subseteq client.\text{requests} \quad \text{(ES-Safety)}$$

We now show how the developer applies the Kondo methodology to proving this protocol.

**Step ❶.** The user starts by writing a synchronous version of the Echo Server protocol, $\mathcal{P}_{sync}$. They define the states of hosts (Figure 7), and the transitions that they can make (Figure 8).

They then define $\mathcal{P}_{sync}$ as the collection of a group of clients and a server. Here, SYNCINIT asserts that every host satisfies their respective initialization predicates:

$$\begin{aligned}\textsc{SyncInit}(S) := \\ & \wedge \textsc{ServerInit}(S.\text{server}) \\ & \wedge \forall\, 0 \leq id < |S.clients| : \textsc{ClientInit}(S.\text{clients}[id], id)\end{aligned}$$

```
datatype Request = Req(clientId: nat, reqId: nat)

datatype Message =
    SubmitReq(req: Request)
    | Response(req: Request)

datatype Client = Variables(
    clientId: nat,                          // unique identifier
    requests: MonotonicSet<Request>,
    responses: set<Request>
)

datatype Server =
    Variables(currentRequest: Option<Request>)

predicate ClientInit(v: Client, id: nat)
    ∧  v.clientId = id
    ∧  v.responses = ∅
    ∧  (∀ req ∈ v.requests : req.clientId = id)

predicate ServerInit(v: Server)
    currentRequest = None
```

Figure 7: Client and server states of the Echo Server protocol, written in Dafny. The MonotonicSet type is wrapper around Dafny's built-in set type and implemented in Kondo's monotonic type library. It indicates to Kondo that the sets are non-decreasing. Meanwhile, ClientInit does not constrain the size of a client's *requests* set, but ensures that every item in *requests* is marked with the client's unique *clientId*.

Meanwhile, SyncNext is a disjunction of two system-level atomic actions:

$$\text{SyncNext}(S, S') := \text{Action}_1(S, S') \vee \text{Action}_2(S, S')$$

In $\text{Action}_1$, a client-server pair performs ClientRequestStep and ServerReceiveStep respectively, where the client sends a request to the server, and the model ensures that the server instantaneously receives the request. In $\text{Action}_2$, a server-client pair performs ServerResponseStep and ClientReceiveStep, with the server sending its response and the client receiving it.

**Step ❷.** Next, the developer writes an inductive proof that $\mathcal{P}_{sync}$ satisfies its safety specification. Here, the inductive invariant is simple. It is the conjunction of ES-Safety with a single predicate asserting that the server's *currentRequest* belongs in the *requests* set of its sender:

$$\forall\, req : server.\text{currentRequest} = \text{Some}(req) \implies req \in clients[req.\text{clientId}].\text{requests} \quad (1)$$

**Step ❸.** Given $\mathcal{P}_{sync}$ written in step ❶, Kondo produces a history-preserving asynchronous version of the Echo Server protocol, $\mathcal{P}_{hist}$.

```
step ClientRequestStep(
    v: Client, v': Client, send: Message)
    ∧ v' = v                          // client state unchanged
    ∧ send.SubmitReq?
    ∧ send.req ∈ v.requests     // send SubmitReq(req)

step ClientReceiveStep(
    v: Client, v': Client, recv: Message)
    // client receives Response

step ServerReceiveStep(
    v: Server, v': Client, recv: Message)
    // server receives Request

step ServerResponseStep(
    v: Server, v': Server, send: Message)
    ∧ v.currentRequest.Some?       // enabling condition
    ∧ v'.currentRequest = None
    ∧  send = Response(v.currentRequest)
```

Figure 8: Client and server transition relations from state $v$ to $v'$, with the bodies of ClientReceiveStep and ServerReceiveStep omitted for brevity. The argument *send* is a new message sent into the network, and *recv* is a message received from the network. Note that the '?' syntax is used to assert if a value is of a particular type or variant.

**Step ❹.** Together with $\mathcal{P}_{hist}$, Kondo also derives a set of Regular Invariants, along with their proof of correctness in $\mathcal{P}_{hist}$. In this example, the only hint required from the user is to label the client's *requests* set as a MonotonicSet type, shown in Figure 7. Note that the client's *responses* set does not need to be labeled as a MonotonicSet, because even though it is monotonic, such a property is not relevant to proving safety. The generated Regular Invariants are:

- A Message Invariant stating that every $\text{Response}(req)$ is such that *req* was once processed by the server:

$$\forall\, \text{Response}(req) \in network : \exists\, i :\ history[i].\text{server.currentRequest} = \text{Some}(req) \quad (2)$$

- A Message Invariant stating that every $\text{SubmitReq}(req)$ is such that *req* is in the *requests* set of the sender:

$$\forall\, \text{SubmitReq}(req) \in network : \exists\, i :\ req \in history[i].\text{clients}[req.\text{clientId}].\text{requests} \quad (3)$$

- A Monotonicity Invariant stating that the *requests* set at each client is non-decreasing:

$$\forall\, i \le j, clientId : history[i].\text{clients}[clientId].\text{requests} \subseteq history[j].\text{clients}[clientId].\text{requests} \quad (4)$$

**Step ❺.** The final step is to prove that $\mathcal{P}_{hist}$ satisfies the safety property. To this end, Kondo generates a draft proof of $\mathcal{P}_{hist}$ by combining the synchronous proof written in step ❶ with the generated Regular Invariants from step ❹. It derives from Equation (1) the history-preserving analogue

$$\forall\, i, req :\ history[i].\text{server.currentRequest} = \text{Some}(req) \implies req \in history[i].\text{clients}[req.\text{clientId}].\text{requests} \quad (5)$$

It then uses the conjunction of Equations (2) to (5) as the protocol's inductive invariant $I$.

**Step ❻.** In this example, the draft proof suffices as the final proof for the asynchronous Echo Server protocol, and no additional effort is required from the user.

### 4.4 Why History Preservation is Important

The Echo Server example also underscores the importance of our history-preservation technique. First, the history-preserving property of $\mathcal{P}_{hist}$ makes deriving and proving Regular Invariants amenable to automation. For instance, observe that for any RESPONSE($req$), its presence in the network implies that the sender of the message performed a SERVERRESPONSESTEP at some point in its execution history, during which $req$ was its *currentRequest* value (Figure 8 line 19). This can be expressed as

$$\begin{aligned}\forall\, msg :\ & msg.\text{RESPONSE?} \wedge msg \in network \\ \implies \exists\, i :\ & \text{SERVERRESPONSESTEP}( \\ & history[i].\text{clients}[msg.\text{src}], \\ & history[i+1].\text{clients}[msg.\text{src}], \\ & msg)\end{aligned} \quad (6)$$

Equation (6) is easy to derive mechanically because it does not contain explicit references to internal host state, yet it implies all the properties that such a step entails, such as Equation (2).

Beyond making Message Invariants easy to derive, history preservation is what allows the invariant taxonomy to apply cleanly to a wide variety of protocols. Consider how the inductive invariant derived in step ❺ of Echo Server proves ES-Safety. First, Equation (2) allows us to relate RESPONSE messages directly to the state of their sender (i.e., the server). Equation (5) then connects the server's state to a prior state of the respective client. Finally, Equation (4) relates that prior state to the current state to imply ES-Safety.

Without preserving history, Monotonicity Invariants such as Equation (4) would be impossible to express. Moreover, any previous values of $server.\text{currentRequest}$ are overwritten and erased from the system, hence preventing us from expressing simple Message Invariants such as Equation (2). Instead, the developer would have to resort to the alternative statement

$$\forall\, \text{RESPONSE}(req) \in network : \text{SUBMITREQ}(req) \in network$$

which mixes the protocol logic of how the server processes requests together with network reasoning, and is neither a Protocol Invariant nor a Regular Invariant. As explained in Section 2.1, this would result in proofs that are less tractable.

## 5 Automation in Kondo

Given a file describing a synchronous protocol $\mathcal{P}_{sync}$, Kondo generates the asynchronous protocol $\mathcal{P}_{hist}$, and human-readable files stating the derived Regular Invariants together with the proof that they are indeed invariants in $\mathcal{P}_{hist}$. Kondo also produces a draft proof of $\mathcal{P}_{hist}$ by combining the user-written synchronous proof with the derived Regular Invariants. In this section, we describe how Kondo derives and proves Regular Invariants with minimal user guidance, and how Kondo generates the draft proof of $\mathcal{P}_{hist}$.

We use the Dafny language and verifier [24] to specify and verify our protocols. We also implement Kondo as a new feature [1] inside the Dafny compiler.

### 5.1 Automating Regular Invariants

**Message Invariants.** In Kondo, we use the special sum type `Message` to define the messages of the system (e.g., Figure 7 line 2). We also require that messages be tagged with the unique identifier of their source host, accessed via $msg$.src. Without loss of generality, we assume that for each message variant $\alpha$, there is exactly one host step $T_\alpha$ that sends that message. If there is more than one, $\alpha$ can simply be split into multiple variants, one for each host step that sends $\alpha$.

Recall that Message Invariants relate the state of hosts to the state of the network via *Send Invariants* and *Receive Invariants* (Section 3.1). Kondo derives Send Invariants asserting that for each message in the network, its sender must have performed the action that sent that message. To do so, Kondo enumerates over `Message` variants. For each variant $\alpha$, Kondo produces the statement that for each $\alpha$ message, $m_\alpha$, in the network, there is some index $i$ in the execution *history* when the source host of $m_\alpha$ performed the step $T_\alpha$ that sent $m_\alpha$ (exemplified by Equation (6)). Such statements yield two critical pieces of information—one, the *enabling condition* of step $T_\alpha$ (i.e., the preconditions required for step $T_\alpha$ to be taken) was satisfied at time $i$; and two, the state at time $i+1$ is in accordance with the transition. When combined with Monotonicity Invariants, they provide information on the current state of the system.

On the other hand, Receive Invariants derived by Kondo assert that if some witness condition $q_\alpha$ is met at a host state $\sigma$, then there must be a message of variant $\alpha$ in the network that made $q_\alpha(\sigma)$ true. More formally, if a host satisfies $q_\alpha$ at index $j$ in its execution history, then there must be an earlier index $i < j$ and message $m_\alpha$ such that $q_\alpha$ is satisfied at index $i+1$ but not $i$, and this step involved the receipt of $m_\alpha$. An example of a witness condition and the derived Receive Invariant for

```
    // User-provided witness for PROMISE messages in Paxos
 1: predicate PROMISEQ(v: ProposerHost, acc: AcceptorId)
 2:     acc ∈ v.promisesSet
 3:
    // Generated Receive Invariant
 4: predicate RECVPROMISEVALIDITY(s: GlobalState)
 5:     ∀ ℓ, j, acc :
 6:       PROMISEQ(s.history[j].proposers[ℓ], acc)
 7:       ⟹
 8:       (∃ i, msg : i < j
 9:         ∧ ¬ PROMISEQ(s.history[i].proposers[ℓ], acc)
10:         ∧ PROMISEQ(s.history[i+1].proposers[ℓ], acc)
11:         ∧ RECVPROMISESTEP(
12:             s.history[i].proposers[ℓ],
13:             s.history[i+1].proposers[ℓ],
14:             msg ) )
15:
```

Figure 9: PROMISEQ is an example of a witness condition for the Paxos protocol. From this definition Kondo generates a corresponding Receive Invariant.

the Paxos protocol is listed in Figure 9. Receive Invariants inform the state of the network given the state of recipient hosts. When combined with Send Invariants, they bridge the relationship between senders and recipients.

Presently, Kondo requires the user to manually write the witness conditions. Kondo then generates one Receive Invariant for each condition. In practice, these are simple conditions that do not require much creativity. For instance, a representative condition is PROMISEQ in the Paxos protocol (Figure 9), which hints that any acceptor's ID in the *promises* set of a proposer host must have arrived via a message. The fact that specific fields in the host state are designed to track information received from messages must already be known by the developer at the time the protocol is conceived.

**Monotonicity Invariants.** These assert the monotonic policies of common data fields in local host state, such as round numbers and write-once variables used to store consensus decisions. In Kondo, we implement a library of common data types, each of which has a partial order relation, *less-than-or-equal-to* (*lteq*). The library includes write-once option types, grow-only numeric types, and append-only sets and sequences. Developers can easily expand this library with custom data types that implement the *lteq* interface.

Whenever the developer uses a monotonic type, Kondo produces an invariant stating that at any point in history, a value of that type must be *lteq* any future value. Equation (4) is one example, where *lteq* is the $\subseteq$ relation for sets. Importantly, the verifier enforces that these types are used correctly.

```
    // User-provided label
 1: predicate HOSTOWNSRESOURCE(v: Host)
 2:     v.hasLock                                  // boolean flag
 3:
    // User-provided label
 4: predicate INFLIGHTFORHOST(v: Host, msg: Message)
 5:     msg.dst = v.hostId  ∧  v.epoch < msg.epoch
 6:
    // Generated Ownership Invariant
 7: predicate ATMOSTONEOWNERPERRESOURCE(
 8:   s: GlobalState)
 9:     ∀ h1, h2 : (
10:       ∧ HOSTOWNSRESOURCE(s.hosts[h1])
11:       ∧ HOSTOWNSRESOURCE(s.hosts[h2])
12:       ⟹ h1 = h2 )
13:
    // Generated Ownership Invariant
    // RESOURCEINFLIGHTBYMSG is auto-generated wrapper
    // around INFLIGHTFORHOST
14: predicate ATMOSTONEINFLIGHT(s: GlobalState)
15:     ∀ m1, m2 : (
16:       ∧ RESOURCEINFLIGHTBYMSG(s, m1)
17:       ∧ RESOURCEINFLIGHTBYMSG(s, m2)
18:       ⟹ m1 = m2 )
19:
    // Generated Ownership Invariant
    // RESOURCEINFLIGHT is auto-generated wrapper
    // around INFLIGHTFORHOST
20: predicate HOSTOWNSRESOURCEIMPLIESNOTINFLIGHT(
21:   s: GlobalState)
22:     RESOURCEINFLIGHT(s)
23:         ⟹ NOHOSTOWNSRESOURCE(s)
24:
```

Figure 10: Example of user-provided ownership labels and the Ownership Invariants that Kondo generates for the Distributed Lock protocol.

**Ownership Invariants.** Many distributed protocols, such as lock services and sharded stores, revolve around ownership of exclusive resources. Though resources vary greatly in function and behavior, the semantics of what it means for the resource to be uniquely-owned is common. Figure 10 shows an example for how Ownership Invariants work in the Distributed Lock protocol [14], where hosts pass around a unique lock in a ring configuration.

For Kondo to generate Ownership Invariants, the user labels a data type as a uniquely-owned resource and describes its ownership semantics. This is done by defining two predicates under special names. First, HOSTOWNSRESOURCE states what it means for the host to own the lock. Second, INFLIGHTFORHOST describes the enabling condition for a

host to receive a lock that is *in-flight*, meaning that the lock is in transit as a network message and can be received by the destination host. In Distributed Lock, HOSTOWNSRESOURCE says that a host owns the lock when its *hasLock* field is true, and INFLIGHTFORHOST evaluates to true when the message's *epoch* value is larger than that of the host.

We emphasize that these labeled conditions are not new concepts that a user must invent for Kondo. Instead, these are formulas that they must inevitably write for any ownership-based protocol. Kondo just requires them to be named in a standardized format.

Using the two predicates, Kondo generates invariants to cover the semantics of a uniquely owned resource. They assert that at most one host can own the resource, at most one copy of the resource can be in-flight, and that if the resource is in-flight then no one can own the resource in the meantime (respectively, ATMOSTONEOWNERPERRESOURCE, ATMOSTONEINFLIGHT, and HOSTOWNSRESOURCEIMPLIESNOTINFLIGHT).

**Inductive Proofs of Regular Invariants.** All of the Regular Invariants generated by Kondo come with verified Dafny lemmas proving that they are indeed invariants. The systematic structure of Regular Invariants ensures that these lemmas can be derived through simple syntax-driven rules. These invariants are such that every individual Message Invariant and Monotonicity Invariant is self-inductive, while the conjunction of Ownership Invariants is inductive.

## 5.2 Automating the Draft Proof

The final goal is to prove that the history-preserving asynchronous protocol $\mathcal{P}_{hist}$ satisfies its safety property. To aid the user in doing so, Kondo generates a draft proof based on the proof of the synchronous version $\mathcal{P}_{sync}$.

First, Kondo lifts the inductive invariants of $\mathcal{P}_{sync}$ to the history-preserving asynchronous world using the following mechanical transformation. Given a $\mathcal{P}_{sync}$ invariant $I(S)$, its history-preserving analogue is $I'(S_{hist}) := \forall i : I(S_{hist}.history[i])$, which simply states that $I$ is true at all points in the history of $S_{hist}$. An example of this is the transformation of Equation (1) into Equation (5).

Second, Kondo also lifts lemmas for $\mathcal{P}_{sync}$ into lemmas for $\mathcal{P}_{hist}$. It converts inductive proofs of synchronous invariants to those of their history-preserving analogues by adding quantifiers over histories and choosing the appropriate elements in the histories as arguments to functions such as $I$.

One important detail is how Kondo handles *triggers* [29]. Triggers are syntactic patterns involving quantified variables that tell the Dafny verifier when to consider concrete instantiations of the quantifiers. For each quantified formula, the user may either manually supply its triggers or rely on Dafny to infer them automatically. In both these cases, Kondo lifts the triggers used in the synchronous invariants into their asynchronous counterparts. This ensures that the Dafny verifier triggers on the same terms in the draft proof as in the synchronous proof, such that lemmas that pass the verifier for $\mathcal{P}_{sync}$ are also likely to pass for $\mathcal{P}_{hist}$.

Overall, Kondo's generated draft proof is a best-effort syntax-guided translation. As such, it may contain lemmas that do not pass the Dafny verifier. Failing lemmas fall into one of two categories. First, the lemma's body contains $\mathcal{P}_{sync}$-specific constructs, namely the concrete instantiation of synchronous actions that cannot be easily translated into the $\mathcal{P}_{hist}$ context. In these cases, Kondo removes these lines from the $\mathcal{P}_{hist}$ proof, and requires the user to complete the translation. Second, Dafny simply needs more proof annotations to guide the verifier in exploring $\mathcal{P}_{hist}$'s more complex state space. In both these cases, the user embellishes the lemma body with more proof code, and may additionally call upon the generated Regular Invariants. With the candidate inductive invariant already in place, however, the effort demanded in this step is mechanical, and the bulk of user creativity is confined to the initial $\mathcal{P}_{sync}$ proof. We quantify this effort in Section 6.4.

# 6 Evaluation

We evaluate the Kondo methodology and tool by applying it to a wide range of distributed protocols. Informed by our experience in building and verifying distributed systems, this selection seeks to cover the space of common protocols as much as possible. The result is the list of protocols in Table 1, which concern a variety of application domains, ranging from consensus to mutual exclusion and concurrency control. We used Two-Phase Commit, Ring Leader Election, and Lock Server to develop and refine the Kondo approach. We then applied the Kondo approach to the remaining protocols.

Our evaluation determines whether Kondo is effective in helping developers prove the correctness of their distributed protocols. In doing so, we answer the following questions.

1. How applicable is the Kondo methodology in finding the inductive invariants of various distributed protocols? (Section 6.2)
2. How effective is Kondo in reducing the number of invariants a user must derive manually? (Section 6.3)
3. How burdensome is writing proof annotations when using Kondo? (Section 6.4)
4. Are there cases in which the Kondo methodology fails? (Section 6.5)

## 6.1 Evaluation Methodology

Each protocol in Table 1 is described as a state machine in Dafny following the IronFleet style [14], together with an associated safety property. For each protocol, the desired output is a theorem checked by Dafny which shows that the

| | $\mathcal{P}_{sync}$ LoC | Invariant Clauses | | | | | Lines of Proof Code | | |
|---|---|---|---|---|---|---|---|---|---|
| | | No Kondo | Sync | Owner | Mono | Msg | No Kondo | Sync | Mods |
| Echo Server | 260 | 5 | 1 | 0 | 1 | 3 | 93 | 40 | 0 |
| Ring Leader Election [5] | 183 | 6 | 1 | 0 | 0 | 2 | 191 | 56 | 0 |
| Simplified Leader Election [39] | 255 | 7 | 3 | 0 | 1 | 2 | 136 | 94 | 0 |
| Two-Phase Commit | 400 | 8 | 4 | 0 | 1 | 3 | 184 | 133 | 19 |
| Paxos [23] | 631 | 27 | 20 | 0 | 5 | 6 | 856 | 557 | 220 |
| Flexible Paxos [16] | 633 | 27 | 20 | 0 | 5 | 6 | 856 | 554 | 226 |
| Distributed Lock [14] | 194 | 2 | 0 | 3 | 0 | 0 | 64 | 31 | 0 |
| ShardedKV | 213 | 2 | 0 | 3 | 0 | 0 | 172 | 61 | 7 |
| ShardedKV-Batched | 225 | 2 | 0 | 3 | 0 | 0 | 172 | 31 | 0 |
| Lock Server [26] | 287 | 7 | 1 | 6 | 0 | 0 | 267 | 44 | 15 |

Table 1: Summary of proof effort using Kondo. Column '$\mathcal{P}_{sync}$ LoC' is the lines of code of the $\mathcal{P}_{sync}$ protocol description. Under 'Invariant Clauses', 'No Kondo' is the number of invariants a user writes to prove the asynchronous protocol when not using Kondo, while 'Sync' is the number of Protocol Invariants the user writes in completing the synchronous proof when using Kondo. The columns 'Owner', 'Mono' and 'Msg' count the Ownership, Monotonicity and Message Invariants Kondo generates. Under 'Lines of Proof Code', 'No Kondo' is the amount of code a user writes to prove the asynchronous protocol when not using Kondo. Meanwhile, 'Sync' is the amount of user-written code to prove $\mathcal{P}_{sync}$ in Kondo, and 'Mods' represent the total size of lemmas that the user had to modify in completing the draft proof generated by Kondo.

protocol's safety property is an invariant. In particular, we note that every protocol and its respective inductive invariant is outside of EPR.

To obtain the proof of each protocol, we apply the Kondo methodology by first finding and proving the inductive invariant for a synchronous version of the protocol $\mathcal{P}_{sync}$. From $\mathcal{P}_{sync}$ and its proof, the Kondo tool generates the asynchronous protocol $\mathcal{P}_{hist}$, a set of Regular Invariants, and a draft proof of $\mathcal{P}_{hist}$. Finally, we manually fill in any gaps in the draft proof to complete the final output.

## 6.2 Applicability of Kondo and the Invariant Taxonomy

We report that the Kondo methodology succeeds in producing inductive invariants for all 10 protocols. Table 1 tallies how the invariant clauses in each inductive invariant are classified in the invariant taxonomy. We note that the Regular Invariant columns in Table 1 only counts those that are useful in the final inductive invariant; i.e., removing such an invariant causes the proof to fail. In all of the examples we tested, the number of extraneous Regular Invariants generated by Kondo is small (at most 2). Hence, there is little threat of the developer being overwhelmed by a deluge of unneeded invariants.

Overall, this result demonstrates the applicability of Kondo along three fronts. First, it shows that the taxonomy is comprehensive in the properties that it covers. Using only invariants that fall within the taxonomic categories, we are able to express all the properties needed to form the inductive invariants of an extensive set of distributed protocols.

Second, the result shows that the inductive invariant for each of these protocols can be formulated in way that respects the classification between Protocol Invariants and Regular Invariants, where host-level reasoning is cleanly confined to Protocol Invariants. This supports the main hypothesis of the invariant taxonomy—it is the well-delineated core of Protocol Invariants that capture the deeper intuitions of the system design. Beyond this core, a set of easily derivable Regular Invariants completes the rest of the proof.

Third, the result supports the conclusion that Kondo is a viable strategy and tool for proof developers. By using only Protocol Invariants derived from $\mathcal{P}_{sync}$ in conjunction with the Regular Invariants generated using Kondo, we are able to find inductive invariants for a wide selection of protocols.

## 6.3 Reducing the Invariant-Finding Burden

In this study, we investigate the degree to which Kondo alleviates the developer's burden through reducing the number of invariant clauses they need to find manually for a protocol. In the case of Kondo, manually-derived invariants are the Protocol Invariants listed under 'Sync' in Table 1. We compare this to the number of invariants one must devise using the conventional IronFleet approach, listed under 'No Kondo'—these numbers are obtained from performing fully manual proofs of the asynchronous but non-history-preserving versions of the protocols, which represent the conventional way to write these protocols [14, 15].

In most cases, the number of manual invariants is drastically reduced when using Kondo, e.g., by up to 6x for Ring Leader

Election. Surprisingly, for Distributed Lock, ShardedKV and ShardedKV-Batched, the user need not write any invariants at all when using Kondo. This is because these protocols are about managing unique resource ownership, a problem that is trivial in the synchronous model where resources are passed atomically between hosts and there is no need to guard against duplication that may occur in an asynchronous network. In the asynchronous model, Ownership reasoning is in turn handled automatically by Kondo's Ownership Invariants.

For Paxos and Flexible Paxos, we observe that the reduction is less drastic, from 27 to 20 in both cases. An experimental factor contributes to the smaller difference. In the non-history-preserving version of these protocols (used to obtain the 'No Kondo' numbers), we used a simpler state machine for the proposer hosts. In particular, they include a step that received a message and sent the response in the same step, a simplification that made the proofs more tractable. Kondo, however, does not allow steps that perform both a send and a receive (see "Synchronous Protocol" in Section 4.2). Hence, we applied Kondo to a modified proposer state machine where that step was decomposed into two separate steps. Indeed, the fact that Kondo required fewer manual invariants despite this added complication highlights the usefulness of Kondo.

Furthermore, these numbers do not fully reflect the qualitative relief Kondo gives to the developer. Because Protocol Invariants are derived from the synchronous protocol model $\mathcal{P}_{sync}$, the developer can ignore the complications caused by network asynchrony when conceiving them. This luxury makes deriving Protocol Invariants qualitatively easier than the invariants in the traditional setting that is tarnished by the asynchronous network.

Overall, we find that Kondo allows the developer to be responsible for both fewer and simpler invariants across a variety of protocols.

## 6.4 Proof Experience

Because our protocols are not expressed in a decidable logic such as EPR, the user is inevitably tasked with writing proof annotations to convince the verifier of the correctness of the inductive invariant. In the Dafny language, these proof annotations come in the form of lemmas that resemble a hand-written inductiveness proof. Using the Kondo methodology, the user is responsible for writing proof annotations in two steps of the process (steps ❷ and ❻ in Figure 6, respectively):

1. Prove that the conjunction of Protocol Invariants is an inductive invariant of the synchronous protocol $\mathcal{P}_{sync}$.
2. Complete the draft proof of the asynchronous protocol, if it is not already complete. This involves adding proof annotations to the *bodies* of lemmas that fail to verify. Notably, the user need not introduce new lemmas, or modify the pre- and post-conditions of existing lemmas.

The columns 'Sync' and 'Mods' in Table 1 under the heading 'Lines of Proof Code' quantify the above efforts respectively, using lines of code as a proxy. They are in contrast to the 'No Kondo' column, which represents the amount of proof a user writes when not using Kondo.

Note that in the 'Mods' column, the numbers represent the total lines of lemmas that required additional proof annotations. In other words, even if just one line had to be added to the lemma, the lines of the entire lemma definition are counted. This is to include conservatively the effort the user may spend reading the lemma in order to complete the proof.

Finally, we emphasize that completing the asynchronous draft proof is a mechanical process once the developer has the synchronous proof in place. In particular, in the asynchronous proof, the developer uses exactly those lemmas already defined in the sync-proof, and the logical reasoning behind why the lemmas are true remains identical. The developer simply adds more assertions in the proof to guide the verifier in exploring a larger state space.

## 6.5 Limitations

Kondo targets safety proofs of crash fault tolerant distributed protocols. As such, liveness proofs are beyond its scope. Moreover, Kondo is not applicable in a Byzantine fault model, as it is not safe to relate a message to the state of a Byzantine-faulty sender or receiver.

Next, Kondo is not guaranteed to be complete, in that it may not work for every protocol or safety property. First, Protocol Invariants derived based on the synchronous protocol may not be correct invariants in the asynchronous model. For instance, the property "there is at least one node holding the lock" is an invariant in the synchronous version of Distributed Lock, but not the asynchronous one where the lock can be in-flight and not held by any node. In our evaluation, however, we have not encountered any safety properties where such invariants were required.

Second, even when the Protocol Invariants derived in the synchronous protocol are correct invariants in the asynchronous version, it is not theoretically guaranteed that their conjunction with Kondo-generated Regular Invariants must be an inductive invariant of the asynchronous protocol. However, we did not come across any such examples.

Ultimately, unlike EPR-based techniques, Kondo is intended to augment, rather than replace, general verification frameworks. In cases where Kondo fails to apply, it is always possible to fall back to the general verification framework, albeit giving up on the full benefits of using Kondo's structured invariants. It is also possible that new invariant categories and techniques may be needed as we apply Kondo to more diverse protocols. In this sense, Kondo comprises a toolkit of techniques that is general enough to cover a useful set of protocols, and it may grow to accommodate new classes of problems as they arise.

## 7 Related Work

Verification of distributed systems has received substantial attention, with proposed solutions falling along a spectrum of automation with differing trade-offs.

**Manual Proofs.** IronFleet's proof of Paxos [14] and Verdi's proof of Raft [42] both use general-purpose theorem provers to tackle their respective correctness proofs. They require entirely handwritten invariants and proofs, cumulating in 4,581 lines of Dafny for IronFleet and 50,000 lines of Coq for Verdi. However, they both accomplish the feat of verifying not just the protocol in the abstract, but an entire executable implementation, whereas Kondo is concerned with the protocol.

**Automated Invariant Inference.** To reduce the substantial effort needed for these proofs, considerable work has focused on automating the process. Ivy [32] makes use of the effectively propositional logic fragment (EPR [34]), which makes inductiveness-checking decidable and efficient in practice. Building on this, several algorithms aim to automate the construction of invariants wholesale; these include I4 [26], SWISS [13], DistAI [44], IC3PO [11], DuoAI [43], Primal-Dual Houdini [33], and P-FOL-IC3 [19].

However, EPR restricts how developers can express their protocols. In invariant inference, this restriction applies to the protocol description and its inductive invariant. For example, to handle Paxos, Padon et al. [31] develop abstractions that transform the verification conditions into EPR, a creative process aided by knowing the invariants in advance. Meanwhile, most approaches that can automatically infer invariants for Paxos (e.g., SWISS) require the protocol to already be transformed. Kondo, however, is not limited to EPR, so it does not share these restrictions. This is possible because it is not a *fully* automatic approach, instead allowing some human intervention. As a result, our solution to Paxos uses a natural, non-transformed protocol description.

Other works that infer invariants outside of EPR include a paper [12] targeting Paxos and the endive tool [36], which verifies a Raft-based protocol using the expressive language of TLA+. Being outside EPR, they cannot check invariants automatically in the unbounded domain, similar to Kondo. The endive authors report providing human guidance but did not quantify such effort.

**Leveraging the Structure of Distributed Systems.** Like Kondo, some work has used the observation that synchronous systems are easier to reason about than asynchronous ones. Pretend Synchrony [40], for example, rewrites asynchronous protocols into synchronous ones with much simpler invariants. However, it requires protocols to obey a restriction called "round non-interference" which precludes certain optimizations that save state between rounds, as in Multi-Paxos.

Some work introduces reasoning principles for the round-based *Heard-Of model* [6], including PSync [9], the $\mathbb{CL}$ logic [8], and *ConsL* [27]. Of course, these frameworks are only applicable to protocols that operate in rounds. Other work in this area [7] makes use of the communication-closure property of some protocols, but may not apply to protocols that do not have this property, such as the Echo Server.

Like Kondo, the work on *message chains* [28] identifies a class of useful invariants based on an insight into the structure of distributed systems. It proposes *message-chain invariants* that accumulate history inside network messages, explicitly mixing host and network state. In contrast, Kondo aims to isolate these two concerns.

**Leveraging Ownership.** Frameworks such as Aneris [20] and Grove [38] use separation logic [35] to reason about distributed systems. Separation logic is particularly good at reasoning about resource ownership, a concept also captured by Kondo's Ownership Invariants. Separation logic is very expressive, but it also requires the developer to come up with invariants (a process that is hard to automate), and it often requires significant technical expertise to use effectively.

## 8 Conclusion

This paper presents an invariant taxonomy that identifies structure in the inductive invariants of distributed protocols. The taxonomy classifies invariants into Regular Invariants (with regular structure that follows from the protocol description) and Protocol Invariants (which capture protocol-specific reasons why the protocol is correct). Building on this insight, the Kondo methodology gives developers a workflow and tool for coming up with an inductive invariant and proving inductiveness. They identify the Protocol Invariants on a synchronous version of the protocol; then use the Kondo tool to get an asynchronous protocol description and Regular Invariants; and finally, prove the inductiveness of the conjunction of Protocol and Regular Invariants, by completing the draft proof generated by Kondo.

## Acknowledgment

We thank Andrea Lattuada and Jon Howell for the fruitful early discussions about this project. We also thank our shepherd, Alexey Gotsman, and the anonymous OSDI reviewers for their great feedback. This work was supported by a gift from VMware and by National Science Foundation grants CCF-2318953 and CCF-2318954.

## References


[1] The Kondo tool. Available online at: https://github.com/GLaDOS-Michigan/Kondo.

[2] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2021, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Aaron R. Bradley. Understanding IC3. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, page 1–14, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] Aaron R. Bradley and Zohar Manna. Property-directed incremental invariant generation. *Form. Asp. Comput.*, 20(4–5):379–405, Jul 2008.

[5] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.

[6] Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Computing*, 22, 04 2009.

[7] Andrei Damian, Cezara Drăgoi, Alexandru Militaru, and Josef Widder. Communication-closed asynchronous protocols. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 344–363, Cham, 2019. Springer International Publishing.

[8] Cezara Drăgoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)*, 2014.

[9] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: A partially synchronous language for fault-tolerant distributed algorithms. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[10] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 405–425, Cham, 2019. Springer International Publishing.

[11] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*, page 131–150, Berlin, Heidelberg, 2021. Springer-Verlag.

[12] Aman Goel and Karem A. Sakallah. Towards an automatic proof of Lamport's Paxos. *CoRR*, abs/2108.08796, 2021.

[13] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021.

[14] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2015, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, Jun 2017.

[16] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016.

[17] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *J. ACM*, 64(1), Mar 2017.

[18] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery.

[19] Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. Inferring invariants with quantifier alternations: Taming the search space explosion. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 338–356, Cham, 2022. Springer International Publishing.

[20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars

Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In Peter Müller, editor, *Programming Languages and Systems*, pages 336–365, Cham, 2020. Springer International Publishing.

[21] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[22] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[23] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, Dec 2001.

[24] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. Sift: Using refinement-guided automation to verify complex distributed systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 151–166, Carlsbad, CA, July 2022. USENIX Association.

[26] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2019, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *Proceedings of Computer Aided Verification (CAV)*, 2017.

[28] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. Message chains for distributed system verification. *Proc. ACM Program. Lang.*, 7(OOPSLA2), Oct 2023.

[29] Michał Moskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, page 20–29, New York, NY, USA, 2009. Association for Computing Machinery.

[30] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 2015.

[31] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[32] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2016, page 614–630, New York, NY, USA, 2016. Association for Computing Machinery.

[33] Oded Padon, James R. Wilcox, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken. Induction duality: Primal-dual search for invariants. *Proc. ACM Program. Lang.*, 6(POPL), Jan 2022.

[34] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *Journal of Automated Reasoning*, 44(4):401–424, Apr 2010.

[35] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 55–74, USA, 2002. IEEE Computer Society.

[36] William Schultz, Ian Dardik, and Stavros Tripakis. Plain and simple inductive invariant inference for distributed protocols in TLA+. *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pages 273–283, 2022.

[37] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL), Dec 2017.

[38] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: A separation-logic library for verifying distributed systems. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2023, page 113–129, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug 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 2018, page 662–677, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: Synchronous verification of asynchronous distributed programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2019.

[41] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas 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 2015, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[42] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *ACM Conference on Certified Programs and Proofs (CPP)*, Jan 2016.

[43] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 485–501, Carlsbad, CA, July 2022. USENIX Association.

[44] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021.

[45] Tony Nuda Zhang, Upamanyu Sharma, and Manos Kapritsos. Performal: Formal verification of latency properties for distributed systems. *Proc. ACM Program. Lang.*, 7(PLDI), Jun 2023.

[46] Naiqian Zheng, Mengqi Liu, Yuxing Xiang, Linjian Song, Dong Li, Feng Han, Nan Wang, Yong Ma, Zhuo Liang, Dennis Cai, Ennan Zhai, Xuanzhe Liu, and Xin Jin. Automated verification of an in-production DNS authoritative engine. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2023, page 80–95, New York, NY, USA, 2023. Association for Computing Machinery.