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

Paper #474

Abstract

Proving the correctness of distributed system protocols is a challenging task. Central to proving that a safety property holds for a protocol is finding an *inductive invariant*. Currently, automated invariant inference algorithms rely on the developer to describe the protocol with a restricted logic. If the developer wants to prove a protocol expressed without these restrictions, they must devise an inductive invariant manually.

In this paper, we propose an approach that simplifies and partially automates identifying 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 structures, and Protocol Invariants, which capture the host relationships that make the protocol work.

Building on the insight of this taxonomy, we describe the Kondo approach for proving correctness of a distributed protocol modeled as a state machine in Dafny. The developer first manually devises the Protocol Invariants by working with a *synchronous* version of the protocol, in which 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 the user combines this output with the proof of the synchronous protocol to finish the invariant proof for the asynchronous protocol. Our evaluation shows that Kondo reduces developer effort for a wide variety of distributed protocols.

1 Introduction

Distributed systems are notoriously difficult to design and reason about. Because they underlie critical applications and infrastructure, any bugs can have severe consequences. Hence, in recent years, both researchers [12, 18, 23, 34, 35, 38, 42] and practitioners [1, 27, 43] have turned to formal verification to rigorously prove the correctness of distributed systems.

At the core of a formal distributed system safety proof is an *inductive invariant*, which implies that a desired safety property holds throughout a system's execution. As argued in previous work [11, 24, 40, 41], manually deriving inductive invariants is a creative challenge. For example, the IronFleet [12, 13] authors reported spending months to identify and prove the inductive invariant of Paxos [20, 21].

Unsurprisingly, this challenge spurred a new category of algorithms and tools to automatically find the inductive invariants of distributed protocols [8, 11, 15–17, 24, 30, 40, 41]. For instance, DuoAI [40] finds the inductive invariant of 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 checking the correctness of inductive invariants is a decidable problem. As such, they apply only to protocols that operate within the confines of a first-order logic fragment known as Effectively Propositional Reasoning (EPR [31]). For example, EPR does not permit arithmetic, thus requiring creative abstractions to encode common systems primitives such as epoch numbers and vote counting. As detailed by Padon et al. [28], expressing a protocol like Paxos in EPR is difficult.

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

In this work, we present a new approach to bridge the gap between the categories above. 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 strata can be derived almost fully automatically, even in a non-EPR setting that permits intuitive programming constructs such as arithmetic, sets and maps. Interestingly, proving the top-most stratum is often equivalent to proving the safety of a simpler, synchronous version of the protocol.

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 objects (*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 the system 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 fundamental building blocks for proving Protocol Invariants. This 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 and proven automatically given the protocol description, even in a general purpose verification tool such as Dafny [22]. We observe that these automatically derived Regular Invariants, in conjunction with a set of user crafted Protocol Invariants, suffice in proving a wide variety of distributed protocols. In fact, these Protocol Invariants are typically 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 to let developers harness the structure afforded by the invariant taxonomy. Using Kondo, the developer focuses her efforts on proving a simpler, synchronous version of the protocol. Kondo generates the asynchronous protocol from the synchronous description, and automatically devises and proves Regular Invariants for the asynchronous protocol. Meanwhile Protocol Invariants are simply carried over from the synchronous proof. The user completes the final proof using only Protocol Invariants and these Regular Invariants.

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 and automation afforded by the invariant taxonomy, and implement the Kondo tool 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 find that Kondo can simplify finding and proving the inductive invariants of these protocols. (Section 6).

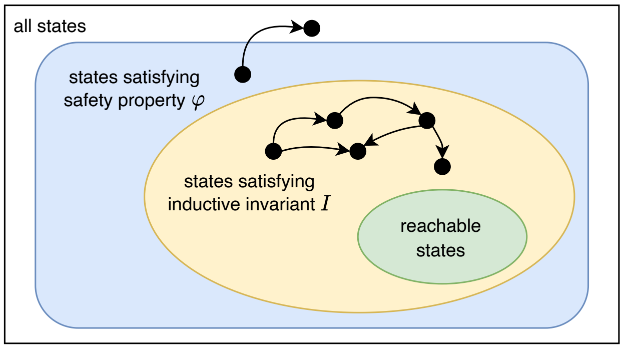
all states
states satisfying safety property φ
states satisfying inductive invariant I
reachable states

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

2 The Challenge of Inductive Invariants

Manually proving the correctness of an asynchronous distributed protocol is widely regarded as a challenging task. This section shows how its difficulty is compounded by the lack of principled techniques for structuring the invariants that a developer must derive for the proof. It results in invariants that entangle complex protocol logic with otherwise standard network semantics. Such invariants require more creativity in reasoning, and can complicate the entire proof.

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

Coming up with the 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 protocol. In distributed systems, this challenge is magnified by the presence of an asynchronous network that may arbitrarily delay, drop, 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 state [12, 26, 28].

2.1 Case Study: Two-Phase Commit

To highlight the challenge in finding inductive invariants, we use the classic two-phase commit protocol (Figure 2). It is parameterized by an arbitrary number of participants,

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

Figure 2: Two-phase commit protocol host and message states. Note that Vote, Decision and Message are sum types.

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. 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.

and has a single coordinator. Participants are initialized with some preference of Yes or No, and they communicate their preferences to the coordinator using the protocol listed in Figure 3. The safety specification we target in this case study is that if any participant reaches a Commit decision, every participant's local *preference* must be Yes:

$$\forall h\,.\; hosts[h].\text{decision} = Some(\text{Commit}) \implies \left(\forall h'.\; hosts[h'].\text{preference} = \text{Yes}\right) \quad \text{(2PC-Safety)}$$

Unfortunately, there is currently no established methodology a developer can follow in finding an inductive invariant for this specification. Instead, she must rely solely on her wit and will, in a journey of intuition-guided improvisation. In particular, she devises a candidate list of other protocol properties that when taken in conjunction with 2PC-Safety, she suspects, will form an inductive invariant.

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

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

Despite its apparent simplicity, proving this invariant requires a non-trivial supporting inductive invariant: 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. The idea of using simple invariants as building blocks for proving more complex ones is not new [2, 3]. We aim to supply a systematic way to apply this idea to distributed systems 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 (e.g., network) invariants. This taxonomy has two distinct categories—an upper strata of Protocol Invariants, and a lower strata of Regular Invariants, illustrated in Figure 4. Regular Invariants are self-inductive and easy to devise, and then 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-state invariants—the portion that demands user creativity—inside a well-demarcated portion of the proof. In turn, the Regular Invariants supporting these invariants are uncontaminated by host-level logic (and in fact automatically derivable; see Section 5).

These categories do not cover the space of all invariants—for a given protocol, there can exist invariants that is neither a Regular Invariant nor a Protocol Invariant. 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 follow from the protocol's steps, without requiring an understanding of why the protocol works. They concern low-level properties that follow from standard network semantics, 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.

We identify the following three subcategories of Regular Invariants: Message Invariants, Monotonicity Invariants, and

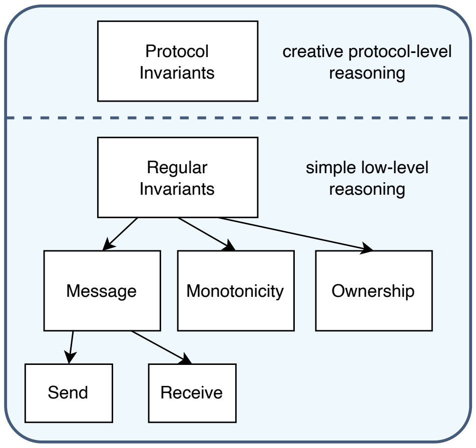

Figure 4: The invariant taxonomy.

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; i.e.,
$$\forall m.\ m \in \mathit{network} \implies p(m, \mathit{hosts}[m.\mathrm{src}])$$
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; i.e.,
$$\forall h.\ q(\mathit{hosts}[h]) \implies \big(\exists m.\ r(\mathit{hosts}[h], m) \land h = m.\mathrm{dst}\big)$$

Monotonicity Invariants. Monotonic data types are a common primitive in distributed protocols [35]. Widely-used structures include grow-only epoch counters to filter outdated 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; i,e.,
$$\forall \sigma, \sigma'.\ \mathit{lt}(\sigma, \sigma')$$
where σ is a historical state of host σ', and the lt represents some order relation of data types within the host state.

Monotonicity invariants require referring to the past state of the host, which is generally not part of the distributed system model. In Section 4.2, we describe our history-preservation technique that enables us to systematically transform the protocol into one that preserves history information. The protocol augmented with history information supports stating and proving Monotonicity Invariants, then 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 uniquely owned resources. Specifically, they say that for each unique resource γ, at most one node can 'own' γ at any point in time, and if γ is in transit in the network, then no nodes can have ownership of γ. These properties are common to protocols that deal with resource ownership.

3.2 Protocol Invariants

Protocol Invariants describe a relationship ℓ among hosts, and do not mention the network. That is, they have the form
$$\forall h_1, \ldots, h_n.\ \ell(\mathit{hosts}[h_1], \ldots, \mathit{hosts}[h_n])$$

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 $\mathcal{P}_{\text{sync}}$. This version, $\mathcal{P}_{\text{sync}}$, 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 $\mathcal{P}_{\text{sync}}$ tend to be helpful Protocol Invariants in $\mathcal{P}$. Moreover, these Protocol Invariants, in conjunction with a set of automatically derived Regular Invariants, tend to be all that is needed to prove $\mathcal{P}$ (evaluated in Section 6.1).

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; the remaining mundane Regular Invariants can be dispatched with minimal effort (even automatically; see Section 5).

$A1$ For each VOTEMSG(v, src) in the network, *src* is a valid participant identifier.

$A2$ For each VOTEMSG(v, src) in the network, v reflects the *preference* of *src*.

$A3$ For each DECIDEMSG(d) in the network, d reflects the local *decision* at the coordinator.

$A4$ For each participant that decided Commit, DECIDEMSG(Commit) must be a message in the network.

$A5$ For each *id* in *yesVotes* at the coordinator, the participant identified by *id* must have the corresponding *preference*.

$A6$ 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. $A5$ and $A6$ are Protocol Invariants, while the rest are Message Invariants.

Two-Phase Commit Revisited. We now revisit the two-phase commit example (introduced in Section 2.1) to demonstrate how the developer can use the invariant taxonomy to bring order and simplicity to her proof. Figure 5 lists a set of invariants for two-phase commit. The conjunction of these invariants forms an inductive invariant that proves the safety property 2PC-Safety.

Of the six invariants, only $A5$ and $A6$ are Protocol Invariants. Discovering them involves protocol-level insight about how state between 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, $A3$ follows directly from protocol step 4. Likewise, $A4$ 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, $A1$ 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. She can first write down the Regular Invariants $A1$, $A2$, $A3$ and $A4$ without any thought about overall protocol correctness. This is easy because these invariants are trivially true just from looking at individual, local protocol steps. Their self-inductive nature also allows the developer to quickly check if the invariants she wrote are correct. Finally, with these Regular Invariants effortlessly in place, the developer then devises the crowning jewels $A5$ and $A6$ 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. Compared to EPR-based approaches (Section 7), Kondo targets general protocol models where proving 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 (Section 5), 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 $\mathcal{P}_{\text{sync}}$ and used directly in the asynchronous protocol $\mathcal{P}^h$. In $\mathcal{P}_{\text{sync}}$, messages are delivered instantaneously between hosts, making it much easier to iteratively devise an invariant and prove inductiveness. In all the distributed protocols in our evaluation (Section 6.1), the Protocol Invariants taken from $\mathcal{P}_{\text{sync}}$, in conjunction with the set of automatically derived Regular Invariants for $\mathcal{P}$, are sufficient to form an inductive invariant for each protocol's safety property.[1]

These observations inform the Kondo methodology, where the developer is responsible for writing and proving a synchronous version of the protocol. Kondo then automatically generates the asynchronous protocol along with a set of Regular Invariants, and with modest effort, the developer lifts her synchronous proof to work for the asynchronous protocol. As a case study, we demonstrate using Kondo to verify a simple Client-Server protocol [40] (Section 4.3).

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 $\mathcal{P}_{\text{sync}}$, 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 $\mathcal{P}_{\text{sync}}$ 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.

Steps ❸, ❹: Given $\mathcal{P}_{\text{sync}}$, the Kondo tool automatically generates an asynchronous protocol $\mathcal{P}^h$ in which hosts communicate through a network that may arbitrarily delay, drop, or re-order messages. $\mathcal{P}^h$ shares the same safety property φ as $\mathcal{P}_{\text{sync}}$. Next, Kondo generates a set of Regular Invariants for $\mathcal{P}^h$, together with lemmas that prove their inductiveness. We denote the conjunction of all Regular Invariants as R. Notably, code generated by Kondo is human-readable.

[1]This is not guaranteed theoretically for all protocols.

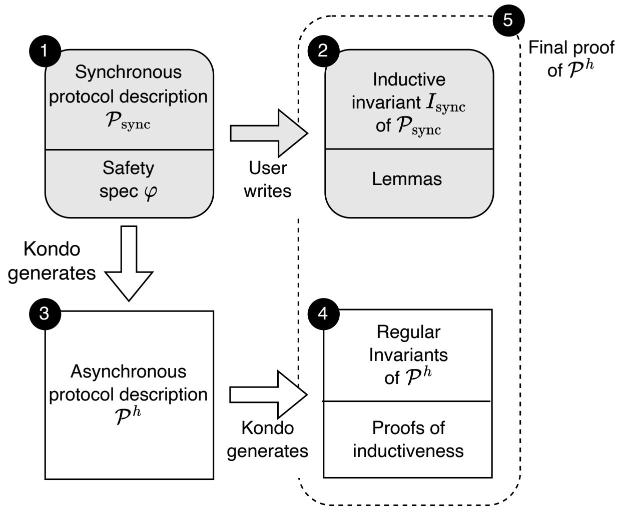
1
Synchronous protocol description $\mathcal{P}_{\text{sync}}$
Safety spec φ
User writes
2
Inductive invariant I_{sync} of $\mathcal{P}_{\text{sync}}$
Lemmas
5
Final proof of $\mathcal{P}^h$
Kondo generates
3
Asynchronous protocol description $\mathcal{P}^h$
Kondo generates
4
Regular Invariants of $\mathcal{P}^h$
Proofs of inductiveness

Figure 6: Kondo workflow. Shaded bubbles represent Dafny files that the user writes. White boxes represent Dafny code and proofs that Kondo generates. The artifacts from ❷ and ❹ are used to manually complete the final proof of $\mathcal{P}^h$.

Step ❺: I_{sync} now serves as the Protocol Invariants of $\mathcal{P}^h$. The conjunction $I_{\text{sync}} \wedge R$ is then used as the final inductive invariant of $\mathcal{P}^h$. The proof written for $\mathcal{P}_{\text{sync}}$ also ports over to $\mathcal{P}^h$ with modest mechanical effort. In particular, *no new lemmas* need to be constructed, and the logical line of reasoning is identical to $\mathcal{P}_{\text{sync}}$. However, the user may need to write additional lines of proof annotations to convince the verifier that the lemmas still hold in the asynchronous protocol.

4.2 Protocol Models

We assume a distributed system where hosts communicate by exchanging messages, and hosts may crash for an indefinite amount of time. Like prior work [11, 12, 24, 40, 41], we use the temporal logic of actions (TLA [19]) 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 Leader, Acceptor and Learner hosts respectively.

Hosts are themselves modeled as event-driven state machines. A host can either take a spontaneous action that may or may not send a message, or an action given some input message. Formally, a host is defined by a HOSTINIT(h: Host) and a HOSTNEXT(h: Host, h': host) predicate. 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: $\mathcal{P}_{\text{sync}}$ is initially written by the developer, whereas $\mathcal{P}^h$ is then derived automatically.

Synchronous Protocol $\mathcal{P}_{\text{sync}}$. The state of the synchronous system $\mathcal{P}_{\text{sync}}$ is $S = (\sigma_1, \dots, \sigma_n)$, an n-tuple of host states in the system. Its initial states are defined by the predicate $\text{SYNCINIT}(S) = (\text{HOSTINIT}(\sigma_1), \dots, \text{HOSTINIT}(\sigma_n))$.

This system can take two kinds of atomic transitions, defined through a SYNCNEXT(S_1, S_2) relation. First, one host may take a local action, i.e., one that doesn't send or receive any data. Second, a pair of hosts may take a corresponding send and receive action simultaneously; i.e., a sender transmits a message that is received instantaneously the recipient.

Note that a consequence of tightly coupling sender and receiver 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 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 [12, 13], 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 6.2).

History-Preserving Asynchronous Protocol $\mathcal{P}^h$. Given the synchronous protocol definition $\mathcal{P}_{\text{sync}}$, Kondo automatically generates a *history-preserving* asynchronous protocol $\mathcal{P}^h$. This model adds to the synchronous model an asynchronous network that may arbitrarily delay, drop, or re-order messages. Like prior work [12, 13, 42], we model this network as a monotonically increasing set of sent messages. When a host sends a message, it is added to this set. A host retrieves an arbitrary message from this set when it calls receive.

Unique to Kondo is the history-preserving aspect of $\mathcal{P}^h$. It maintains a sequence *history* of host snapshots, enabling us to express monotonic properties. Formally, let $history = [S_0, \dots, S_m]$ be a sequence of host state n-tuples, where each entry in *history* is a snapshot of all hosts in the system. Then the system state of $\mathcal{P}^h$ is $S^h = (history, N)$. 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

$$\begin{aligned}\text{INIT}(S^h) := \; & len(history) = 1 \\ & \wedge \; \text{SYNCINIT}(history[0]) \; \wedge \; N = \emptyset\end{aligned}$$

The current state of the system $S^h.curr()$ is the tuple $(last(history), N)$, where $last(s)$ returns the last item in a sequence s. In a system-level transition, a single host may take a local step that does not interact with the network, send a message into the network, or receive an arbitrary message from the network. During such a transition, the latest system states obey the relation $\text{ASYNCNEXT}(S^h_1.curr(), S^h_2.curr())$. The transition of the entire history-preserving state machine

```
datatype Request = Req($clientId$: nat, $reqId$: nat)
datatype Message =
    SubmitReq($req$: Request)
    | Response($req$: Request)
datatype Client = Variables(
    $clientId$: nat,                        // unique identifier
    $requests$: set<Request>,
    $responses$: set<Request>
)
datatype Server =
    Variables($currentRequest$: Option<Request>)
predicate ClientInit($v$: Client, $id$: nat)
    $\wedge$  $v.$clientId $= id$
    $\wedge$  $v.$responses $= \emptyset$
    $\wedge$  ($\forall$ $req \in v.$requests. $req.$clientId $= id$)

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

```

Figure 7: Client and server states of the Client-Server protocol. ClientInit does not constrain the size of a client's *requests* set, but ensures that every *req* in *requests* is marked with the client's unique *clientId*.

```
step ClientRequestStep(
  $v$: Client, $v'$: Client, $send$: Message)
    $\wedge$ $v' = v$                      // client state unchanged
    $\wedge$ $send.$SubmitReq?
    $\wedge$ $send.$req $\in v.$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)
    $\wedge$ $v.$currentRequest.Some?       // enabling condition
    $\wedge$ $v' = v.$(currentRequest := None)
    $\wedge$  $send$ = 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.

is then defined by the relation

$$\begin{aligned}\textsc{Next}(S^h_1, S^h_2) := \ & len(history_1) \geq 1 \\ & \wedge\ history_1 = trunc(history_2) \\ & \wedge\ \textsc{AsyncNext}(S^h_1.curr(), S^h_2.curr())\end{aligned}$$

where $trunc(s)$ yields s with the last item removed.

4.3 Example: Client-Server

To illustrate the Kondo methodology, we apply it to a simple Client-Server system [40]. This system comprises an arbitrary number of clients and a single server. Figure 7 describes the client and server states. Each client maintains a set of *requests*, each defined by its unique client and request identifiers.

Clients send their requests to the server. Upon receiving a request, the server sends a response to the request's sender. When a client receives a response, it stores it in a *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:

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

We now show how the developer follows the Kondo methodology for this protocol.

Step ❶. The user starts by writing a synchronous version of Client-Server. She defines the types of hosts in the system (Figure 7), and the transitions that they can make (Figure 8).

She then defines the synchronous protocol as the collection of a server and a group of clients. Here, SyncInit asserts that every host satisfies their respective Init(p)redicates. SyncNext is a disjunction of two system-level atomic transitions. One, 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. Two, a server-client pair performs ServerResponseStep and ClientReceiveStep, with the server sending its response to the client and the client receiving it.

Step ❷. Next, the developer writes an inductive proof for this synchronous protocol. Here, the inductive invariant is simple. It consists of a single predicate

$$\begin{aligned}&\forall\, req.\ server.\text{currentRequest} = Some(req) \\ &\qquad \Longrightarrow req \in clients[req.\text{clientId}].\text{requests} \qquad (1)\end{aligned}$$

asserting that the server's *currentRequest* belongs in the *requests* set of the client identified by *currentRequest*.clientId.

Step ❸. Given the synchronous protocol from step ❶, Kondo automatically generates a *history-preserving* asynchronous protocol $\mathcal{P}^h$.

Step ❹. Together with the $\mathcal{P}^h$ protocol state machine, Kondo also derives a set of Regular Invariants for $\mathcal{P}^h$ with minimal user guidance (Section 5), together with their proof of correctness. These Regular Invariants are:

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

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

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

$$\forall\, \textsc{Response}(req) \in network\,. \\ \exists\, i.\ req \in history[i].\mathrm{clients}[req.\mathrm{clientId}].\mathrm{requests} \quad (3)$$

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

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

Step ❺. The final step is to prove that $\mathcal{P}^h$ satisfies the safety property. The user lifts the inductive invariants of the synchronous protocol, written in step ❷, to the history-preserving asynchronous world using the following mechanical transformation. Specifically, given a $\mathcal{P}_{\text{sync}}$ invariant $I(s)$, its history-preserving analogue is $\forall i.\ I\big((history[i], N)\big)$, which simply states that I is true at all points in history.

In this example, the analogue of Equation (1) is

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

which serves as the sole Protocol Invariant of Client-Server.

Finally, user attempts to prove the conjunction of Equations (2) to (5) as the protocol's inductive invariant I. The user starts with the same lemmas and proof code she wrote in step ❷, and may additionally call upon the the generated lemmas from step ❹ to convince the verifier that I holds in $\mathcal{P}^h$. With the candidate inductive invariant in place, the effort demanded in this step is mechanical, and the bulk of user creativity is confined to the initial $\mathcal{P}_{\text{sync}}$ proof in step ❷.

4.4 Why History Preservation is Important

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

$$\forall\, msg.\ type(msg) = \textsc{Response} \wedge msg \in network \\ \implies \exists\, i.\ \textsc{ServerResponseStep}(\\ history[i].\mathrm{clients}[msg.\mathrm{src}], \\ history[i+1].\mathrm{clients}[msg.\mathrm{src}], \\ msg) \quad (6)$$

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 Client-Server proves CS-Safety. First, Equation (2) allows us to relate $\textsc{Response}$ messages directly to the state of their sender (i.e., the server). Equation (5) then connects the server's state to a historical state of the respective client. Finally, Equation (4) relates that historical state to the current state to imply CS-Safety.

Without preserving history, monotonicity invariants such as Equation (4) would be impossible to express. Moreover, any past values of $server.\mathrm{currentRequest}$ are 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:

$$\forall\, \textsc{Response}(req) \in network\,. \\ \textsc{SubmitReq}(req) \in network$$

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 Automating Regular Invariants

Given a file describing a synchronous state machine $\mathcal{P}_{\text{sync}}$, Kondo generates the history-preserving protocol $\mathcal{P}^h$, and human-readable files stating the derived Regular Invariants, together with the proof that they are indeed invariants in $\mathcal{P}^h$. In this section, we describe how Kondo derives and proves Regular Invariants with minimal user guidance.

We use the Dafny language and verifier [22] to specify and verify our protocols. We then implement Kondo as a new feature inside the Dafny compiler.

Message Invariants. In Kondo, we use the special sum type `Message` to specify the messages of the system (Figure 7). We also require that messages be tagged with the unique identifier of their source host, accessed via *msg*.src. Without

```
// User-provided witness for PROMISE messages in Paxos
predicate PROMISEQ($v$: LeaderHost, $acc$: AcceptorId)
    $acc \in v.promisesSet$

// Generated receive invariant
predicate RECVPROMISEVALIDITY($s$: SystemState)
    $\forall\, \ell,\ j,\ acc.$
        PROMISEQ($s$.history[$j$].leaders[$\ell$], acc)
        $\Longrightarrow$
        $\big(\exists\, i,\ msg.\ i < j$
          $\wedge\ \neg$ PROMISEQ($s$.history[$i$].leaders[$\ell$], acc)
          $\wedge$ PROMISEQ($s$.history[$i+1$].leaders[$\ell$], acc)
          $\wedge$ RECVPROMISESTEP(
                $s$.history[$i$].leaders[$\ell$],
                $s$.history[$i+1$].leaders[$\ell$],
                msg ) $\big)$

```

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

loss of generality, we assume that for each message variant α, there is exactly one host step T_α that sends that message[2].

Recall that Message Invariants relate the state of hosts to the state of the network, via *send invariants*, and *receive invariants* (Section 3.1). The send invariants derived by Kondo assert that for each message in the network, its sender must have performed the action that sent that message. To do so, Kondo enumerates through `Message` variants. For each variant α, Kondo produces the statement that for each α message, m_α, in the network, there is some index i in the execution *history* when the source host of m_α performed the step T_α that sent m_α (exemplified by Equation (6)). Such statements give two critical pieces of information—one, that at time i, the *enabling condition* of step T_α (i.e., the preconditions required for step T_α to be taken) was satisfied; 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 when some witness condition q_α is met at a host state σ, then there must be a message of variant α in the network that made $q_\alpha(\sigma)$ true. More formally, if a host satisfies q_α at index j in its execution history, then there must be an earlier index $i < j$ and message m_α such that q_α is satisfied at index $i+1$ but not i, and this step involved the receipt of m_α. An example of a witness condition and the derived receive invariant is listed in Figure 9 for the Paxos protocol. Receive invariants inform the state of the network given the state of recipient hosts. When combined with send invariants, they bridge the relationship between sender and receiver hosts.

[2]If there are more than one, α can simply be split into multiple variants, one for each host step that sends α

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 user 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 leader 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 a monotonic type is used, 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 such example, where *lteq* is the $\subseteq$ relation for sets.

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 [12], 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 labels its ownership semantics; e.g., HOSTOWNSRESROUCE states that a host owns the lock only when its *hasLock* field is true.

Next, the user labels the enabling condition for a host to receive the resource as an *in-flight* predicate, where in-flight means that the resource is in transit as a network message, and can be received by the destination host. In Distributed Lock, it means that the message's *epoch* value is larger than that of the host (INFLIGHTFORHOST in Figure 10).

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 labeled in a recognizable format.

Using the two labels, Kondo generates invariants to cover the semantics of a uniquely owned object. 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,

```
// User-provided label
predicate HostOwnsResource($v$: Host)
    $v$.hasLock                                    // boolean flag

// User-provided label
predicate InFlightForHost($v$: Host, $msg$: Message)
    $\wedge$ $msg$.dst $=$ $v$.hostId $\wedge$ $v$.epoch $<$ $msg$.epoch

// Generated ownership invariant
predicate AtMostOwnerPerResource(
  $s$: SystemState)
    $\forall\, r, h_1, h_2$ . (
        $\wedge$ HostOwnsResource($s, h_1, r$)
        $\wedge$ HostOwnsResource($s, h_2, r$)
        $\implies h_1 = h_2$)

// Generated ownership invariant
// ResourceInFlightByMsg is auto-generated wrapper
// around InFlightForHost
predicate AtMostOneInFlight($s$: SystemState)
    $\forall\, r, m_1, m_2$ . (
        $\wedge$ ResourceInFlightByMsg($s, m_1, r$)
        $\wedge$ ResourceInFlightByMsg($s, m_2, r$)
        $\implies m_1 = m_2$)

// Generated ownership invariant
// ResourceInFlight is auto-generated wrapper
// around InFlightForHost
predicate HostOwnsResourceImpliesNotInFlight($s$:
SystemState)
    $\forall\, r$. NoHostOwnR($s, r$)
        $\implies$ $\neg$ResourceInFlight($v, r$)
```

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

AtMostOwnerPerResource, 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 are self-inductive, while the conjunction of Ownership Invariants is inductive.

6 Evaluation

We evaluate Kondo by applying it to a wide range of distributed protocols, listed in Table 1. They 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. In particular, we note that none of these protocol specifications are in EPR. Each protocol and an associated safety property is described as a state machine in Dafny following the IronFleet style [12], and the result of our overall approach is a theorem checked by Dafny which shows the protocol's safety property is an invariant.

Our evaluation determines whether the Kondo methodology and tool 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 and proving inductive invariants of various distributed protocols? (Section 6.1)
2. How effective is Kondo in reducing the number of invariants a user must derive manually? (Section 6.2)
3. How burdensome is writing proof annotations when using Kondo? (Section 6.3)

6.1 Applicability of Kondo and the Invariant Taxonomy

Here, we evaluate whether the Kondo methodology can help the user to find the inductive invariants of wide variety of distributed protocols. In all the protocols evaluated, the user follows the Kondo methodology by first devising the inductive invariant $\mathcal{I}_{\text{sync}}$ for a synchronous version of the protocol $\mathcal{P}_{\text{sync}}$. $\mathcal{I}_{\text{sync}}$ is then used as a Protocol Invariant to prove the asynchronous protocol $\mathcal{P}^h$. We then write a proof in Dafny that $\mathcal{I}_{\text{sync}}$, together with Regular Invariants generated by Kondo, is an inductive invariant for $\mathcal{P}^h$.

We report that the Kondo methodology succeeds in producing the inductive invariants for all 10 protocols. Table 1 lists the detailed tally of 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, the threat of the developer being overwhelmed by a deluge of Regular Invariants is small.

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

	$\mathcal{P}_{\text{sync}}$ LoC	Msg	Mono	Owner	Proto	Traditional	$\mathcal{P}_{\text{sync}}$ proof LoC	$\mathcal{P}^h$ proof LoC
Client-Server [40]	260	3	1	0	1	5	40	53
Ring Leader Election [4]	179	2	0	0	1	6	69	78
Simplified Leader Election [36]	255	2	1	0	3	5	103	143
Two-Phase Commit	394	3	1	0	4	10	139	156
Paxos [21]	629	6	5	0	20	24	597	935
Flexible Paxos [14]	631	6	5	0	20	24	595	929
Distributed Lock [12]	201	0	0	3	0	2	31	58
ShardedKV	213	0	0	3	0	2	59	58
ShardedKV-Batched	225	0	0	3	0	2	38	60
Lock Server [24]	294	0	0	6	0	5	37	59

Table 1: Summary of proof effort using Kondo. The columns 'Msg', 'Mono' and 'Owner' count the Message, Monotonicity and Ownership invariants Kondo generates. 'Proto' is the number of Protocol Invariants the user writes. Columns '$\mathcal{P}_{\text{sync}}$ LoC', '$\mathcal{P}_{\text{sync}}$ proof LoC' and '$\mathcal{P}^h$ proof LoC' are the source lines of code of $\mathcal{P}_{\text{sync}}$ protocol description, $\mathcal{P}_{\text{sync}}$ proof, and $\mathcal{P}^h$ proof respectively. 'Traditional' is the number of invariants a user must write to prove the asynchronous protocol when not using Kondo.

that fall within the taxonomic categories, we are able to express all the properties needed to form the inductive invariant for an extensive set of distributed protocols.

Second, it 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 handles the rest of the proof.

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

6.2 Reducing the Invariant-Finding Burden

In this experiment, we investigate the degree to which Kondo alleviates the developer's burden by reducing the number of invariant clauses she needs to derive manually for a protocol. In the case of Kondo, these are the Protocol Invariants in Table 1. We compare this to the number of invariants she must devise using the conventional IronFleet approach, listed under 'Traditional' in Table 1. These numbers are obtained from performing a fully manual proof of an asynchronous but non-history-preserving version of the protocols, which represent the standard way to write these protocols [12, 13].

In most cases, we see that the number of manual invariants is drastically reduced when using Kondo—up to six-fold for Ring Leader Election. Surprisingly, for Distributed Lock, ShardedKV, ShardedKV-Batched, and Lock-Server, we see that the user need not write any Protocol Invariants at all using Kondo. This is because these protocols are about managing unique resource ownership, and this is handled automatically by Ownership Invariants. Unique ownership is trivial in the synchronous model where the resource is passed atomically between hosts, and there is no need to guard against duplication that may occur in an asynchronous network.

For Paxos and Flexible Paxos, we observe that the reduction is less drastic, from 24 to 20 in both cases. An experimental factor contributes to the smaller difference. In the non-history-preserving version of these two protocols (used to obtain the 'Traditional' numbers), we used a simpler host state machine. In particular, these state machines included a step that received a message and sent the response in the same step, a simplification that made the invariants more tractable. The Kondo methodology, however, does not allow steps that perform both a send and a receive (see "Synchronous Protocols" in Section 4.2). Hence, we applied Kondo to a modified host state machine where that step was broken 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 show the qualitative relief Kondo gives to the developer. Because Protocol Invariants are derived from the synchronous protocol model $\mathcal{P}_{\text{sync}}$, in conceiving them, the developer can ignore the complications caused by network asynchrony. This luxury makes deriving Protocol Invariants qualitatively easier than the invariants in the traditional setting that is tarnished by the asynchronous network. Hence, we conclude that Kondo mitigates the developer's burden when finding inductive invariants.

6.3 Proof Experience

Because our protocols are not expressed in a decidable logic such as EPR, the user is 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. In the Kondo methodology, Kondo completes the proof of Regular Invariants fully automatically. The user is then responsible for writing proof annotations for two steps of the process (steps ❷ and ❺ in Figure 6, respectively):

1. Prove that the conjunction of Protocol Invariants is an inductive invariant of the synchronous protocol $\mathcal{P}_{\text{sync}}$ (call this the sync-proof).
2. Prove that the conjunction of Protocol Invariants and Kondo-generated Regular Invariants is an inductive invariant of the asynchronous protocol $\mathcal{P}^h$ (call this the async-proof).

The final two columns in Table 1 quantify how much mechanical effort the async-proof demands over the sync-proof, using lines of proof code as a proxy.

Proofs are needed because Protocol Invariants capture properties that demand user creativity, making them less amenable to automation. Once the developer has the sync-proof in place, transforming it into the async-proof is mechanical. In particular, in the async-proof, the developer uses exactly those lemmas already defined in the sync-proof, and the logical reasoning behind why these lemmas are true remains identical. The developer simply adds more assertions in the proof to guide the verifier in the right direction.

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 [12] and Verdi's proof of Raft [39] both use general-purpose theorem provers to tackle their respective correctness proofs. They require entirely handwritten invariants and proofs, producing 12,000 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 runnable 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 [29] makes use of the restricted logical fragment *EPR* (effectively propositional reasoning) [31] 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 [24], SWISS [11], DistAI [41], IC3PO [9], DuoAI [40], Primal-Dual Houdini [30], and P-FOL-IC3 [17].

The EPR restriction in invariant inference applies to both the protocol description and the inductive invariants. For example, to handle Paxos, Padon et al. [28] 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 [10] targeting Paxos and the endive tool [33], which verifies a Raft-based protocol using the expressive language of TLA+. Being outside EPR, they cannot check invariants automatically in the unbounded domain, though the endive authors report checking with human guidance, similar to Kondo.

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 [37], 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* [5], including PSync [7], the $\mathbb{CL}$ logic [6], and *ConsL* [25]. Of course, these frameworks are only applicable to protocols that operate in rounds.

Like Kondo, the work on *message chains* [26] identifies a class of useful, structured invariants based on insight into the structure of distributed systems. These invariants, *message-chain invariants*, accumulate history inside network messages, explicitly mixing host and network state, in contrast to Kondo, which aims to isolate these two concerns.

Leveraging Ownership. Frameworks such as Aneris [18] and Grove [35] use separation logic [32] to reason about distributed systems. Separation logic is particularly good at reasoning about 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 to identify structure in a distributed protocol's inductive invariant, classifying invariants into Regular Invariants (with regular structure that follows from the protocol description) and Protocol Invariants (which capture the application-specific reason 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 self-inductive Regular Invariants, and finish by proving the inductiveness of the final invariant.

References

[1] 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 ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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 Symposium on Operating Systems Principles*, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[13] 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.

[14] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited, 2016.

[15] 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.

[16] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. PLDI 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery.

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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 Symposium on Operating Systems Principles*, SOSP '19, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery.

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

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

[27] 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.

[28] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. 1(OOPSLA), oct 2017.

[29] 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 '16, page 614–630, New York, NY, USA, 2016. Association for Computing Machinery.

[30] 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.

[31] 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.

[32] John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.

[33] 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.

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

[35] 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 Symposium on Operating Systems Principles*, SOSP '23, page 113–129, New York, NY, USA, 2023. Association for Computing Machinery.

[36] 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.

[37] 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.

[38] 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 '15, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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 Symposium on Operating Systems Principles*, SOSP '23, page 80–95, New York, NY, USA, 2023. Association for Computing Machinery.