Interactive Safety Verification of Distributed Protocols by Inductive Proof Decomposition

WILLIAM SCHULTZ, Northeastern University, USA
EDWARD ASHTON, Azure Research, Microsoft, UK
HEIDI HOWARD, Azure Research, Microsoft, UK
STAVROS TRIPAKIS, Northeastern University, USA

Many techniques for automated verification of distributed protocols have been developed over the past several years, but their performance is still unpredictable and their failure modes can be opaque for industrial scale verification tasks. Thus, in practice, large-scale verification efforts typically require some amount of human guidance. In this paper, we present *inductive proof decomposition*, a new methodology for interactive safety verification that provides a compositional, interactive approach to inductive invariant development. Our approach guides the human-aided development of inductive invariants via a novel structure, an *inductive proof graph*, which is built incrementally by a human verifier, working backwards from a target safety property. A user is guided by induction counterexamples that are localized to specific nodes of this graph, and nodes of this proof graph are further decomposed based on logical actions that appear in a protocol's transition relation. Our decomposition also enables a novel *slicing* technique that hides irrelevant protocol state at each sub-component of an inductive proof, allowing a user to focus on fine-grained sub-problems rather than a large, monolithic inductive invariant. We present our technique and experience applying it to develop inductive safety proofs of several complex distributed protocols, including the Raft and Zab consensus protocols, which are beyond the capabilities of modern automated verification tools. We also demonstrate how the developed proof graphs provide additional insight into the structure of a protocol proof and its correctness.

1 INTRODUCTION

Verifying the safety of large-scale distributed and concurrent systems remains an important and difficult challenge. These protocols serve as the foundation of many modern fault-tolerant systems, making the correctness of these protocols critical to the reliability of large scale database and cloud systems [8, 18, 43]. Formally verifying the safety of these protocols typically centers around development of an *inductive invariant*, an assertion about system state that is preserved by all protocol transitions. Developing inductive invariants, however, is one of the most challenging aspects of safety verification and has typically required a large amount of human effort for real world protocols [48, 49].

Over the past several years, particularly in the domain of distributed protocol verification, there have been several recent efforts to develop more automated inductive invariant development techniques [11, 26, 41, 51]. Many of these tools are based on modern model checking algorithms like IC3/PDR [11, 12, 24–26], and others based on syntax-guided or enumerative invariant synthesis methods [15, 50]. These techniques have made significant progress on solving various classes of distributed protocols, including some variants of real world protocols like the Paxos consensus protocol [12, 29]. The theoretical complexity limits facing these techniques, however, limit their ability to be fully general [40] and, even in practice, the performance of these tools on complex protocols is still unpredictable, and their failure modes can be opaque. In particular, a key drawback of these methods is that, in their current form, they are very much "all or nothing". That is, if they solve a given problem, then no manual proof effort is needed, but if a problem falls outside the

Authors' addresses: William Schultz, schultz.w@northeastern.edu, Northeastern University, Boston, Massachusetts, USA; Edward Ashton, , Azure Research, Microsoft, Cambridge, UK; Heidi Howard, , Azure Research, Microsoft, Cambridge, UK; Stavros Tripakis, , Northeastern University, Boston, USA.

scope of what they can solve, little assistance is provided in terms of how to develop a manual proof or how a human can offer guidance to the tool.

In practice, real world, large-scale verification efforts often require some amount of human interaction i.e., a human provides guidance when an automated engine is unable to automatically prove certain properties about a design or protocol. For example, recent verification efforts of industrial scale protocols either note the high amount of human effort in developing inductive invariants or leave them as future goals [6, 42]. Several recent, automated approaches have also adopted a paradigm of integrating human assistance to accelerate proofs for larger verification problems e.g., in the form of a manually developed refinement hierarchy [12, 33].

Though there has been a large amount of work on scaling *automated* protocol verification techniques, there has been considerably less focus on *interactive* verification. That is, consideration of how a human can proceed effectively with an inductive proof when a tool fails to solve a verification task automatically. The Ivy framework [41] was a notable, recent attempt to address the interactive safety verification problem, but its techniques were targeted at specific goals which only addressed partial aspects of the problem. Namely, their focus was primarily on (1) ensuring decidability of verification conditions and (2) incorporating a human into the loop of counterexample generalization heuristics. Though this addressed some aspects of the human-machine verification interface, it did not consider other, key issues that arise in large-scale inductive proof efforts. For example, it did not consider how to manage the structure of a large inductive invariant effectively as it is being developed, how to provide feedback to a user about progress on the proof, or how to effectively allow localized reasoning on sub-components of the proof, etc.

In addition to frameworks like Ivy, there is a large amount of work on the use of interactive theorem proving for system verification [7, 16] e.g., using systems like Coq [3], Isabelle/HOL [36], ACL2 [20], etc. The learning curve for these tools is typically steep, though, and they have typically offered a significantly lower degree of automation, making them more laborious to use for many verification efforts and for protocol designers or engineers [35]. Thus, although these tools provide a relatively high degree of interactivity, they are often quite complex and tedious to use for practical verification efforts.

In this paper we present a new, interactive safety verification methodology, *inductive proof decomposition*, that provides a compositional approach to interactive inductive invariant development that allows for a smooth integration between human effort and machine guidance for large-scale safety proof efforts. We are focused on the human-aided development of inductive invariants, typically with the assistance of a backend solver for checking inductive proof obligations that can provide counterexamples to induction. Standard approaches to this process (e.g., as in the paradigm of Ivy) essentially proceed by having a human verifier examine induction counterexamples in a linear fashion, with a goal of constructing a monolithic list of lemma invariants whose conjunction form a valid inductive invariant. We argue that this standard model is poorly suited for large and complex verification efforts, where inductive invariants may grow to include potentially dozens of complex conjuncts about protocol state. Our technique aims to make this process fundamentally *compositional*, which we believe is a core aspect of any "intuitive" proof process undertaken by a human (e.g. as in "pen and paper" style proofs), and is also crucial for managing complexity in any large proof effort [28, 44, 47].

Our technique is based around a novel, formal structure which we define, the *inductive proof graph*, which imposes a particular compositional structure on an inductive invariant, while also taking into account the distinct logical actions that are present in most concurrent and distributed protocols. This graph structure makes explicit the relative induction dependencies between lemmas of a monolithic inductive invariant, and our proof methodology guides a human verifier to incrementally construct this graph structure by working backwards from a target safety property. Throughout the

process, machine guidance is provided in the form of relative induction counterexamples that are maintained at each node of this graph, and so can be reasoned about locally, rather than considered in the scope of the entire inductive invariant. This also allows for local abstractions to be applied that reduce the complexity of counterexamples to be examined. In particular, we present a novel *counterexample slicing* technique, that projects out protocol state variables that are irrelevant to a node of the proof graph, often significantly reducing the scope of information presented to a user, easing their analysis task.

We apply our technique to several distributed protocols, including 2 large, industrial-scale protocol specifications of the Raft [38] and Zab [23] consensus protocols, demonstrating the effectiveness of our technique and its ability to allow human verifiers to work with large proof structures effectively. We also show how the resulting proof graph artifacts can provide further insight into protocol correctness.

In summary, our contributions are as follows:

- Definition and formalization of *inductive proof graphs*, a formal structure representing the logical dependencies between conjuncts of an inductive invariant and actions of a distributed or concurrent protocol. (Section 4)
- *Inductive proof decomposition*, a methodology for large scale interactive safety proofs that is based around the incremental, counterexample-guided construction of an inductive proof graph. (Section 5)
- Implementation of our technique in an interactive verification tool, and an empirical evaluation of our method on several distributed protocols, including large-scale specifications of the Raft [38] and Zab [23] protocols, and analysis of how the resulting proof artifacts provide insights into protocol correctness. (Section 6)

2 OVERVIEW

In this section we present an overview of *inductive proof decomposition*, our proof methodology. We motivate our approach by discussing the limitations of existing inductive invariant development techniques on a running example, and then illustrate the core ideas of our approach on this example.

Running Example: SimpleConsensus. Figure 1 shows a formal specification of an abstract consensus protocol, *SimpleConsensus*, defined as a symbolic transition system. This protocol utilizes a simple leader election scheme to select values, and is parameterized on a set of nodes, *Node*, a set of values to be chosen, *Value*, and *Quorum*, a set of intersecting subsets of *Node*. Nodes can vote at most once for another node to become leader, and once a node garners a quorum of votes it may become leader and decide a value. The top level safety property, *NoConflictingValues*, shown in Figure 2, states that no two differing values can be chosen. The specification of this protocol consists of 6 state variables and 5 distinct protocol actions. Figure 2 shows a complete inductive invariant, *Ind*, for establishing the *NoConflictingValues* safety property, which consists of 8 total conjuncts, along with a subset of definitions for the lemmas in *Ind*.

2.1 Existing Approaches

In a standard interactive safety verification paradigm (e.g., as done in Ivy [41]), the general technique is based on iterative analysis of counterexamples to induction. That is, to develop an inductive invariant such as the one shown in Figure 2, one starts with the top level safety property, *NoConflictingValues*, and generates counterexamples to induction, iteratively developing new lemma invariants to rule out these counterexamples until the overall invariant becomes inductive.

For large scale protocol verification efforts, there are several issues that arise with this basic framework. At a high level, when systems and their invariants become large, it becomes increasingly

CONSTANTS $Node, Value, Quorum$

VARIABLES
$voteRequestMsg, voted,$
$voteMsg, votes,$
$leader, decided$

$Init \triangleq$ Initial states.
$\land\ voteRequestMsg = \{\}$
$\land\ voted = [i \in Node \mapsto \text{False}]$
$\land\ voteMsg = \{\}$
$\land\ votes = [i \in Node \mapsto \{\}]$
$\land\ leader = [i \in Node \mapsto \text{False}]$
$\land\ decided = [i \in Node \mapsto \{\}]$

$Next \triangleq$ Transition relation.
$\exists\, i, j \in Node : \exists\, v \in Value :$
$\exists\, Q \in Quorum :$
$\lor\ SendRequestVote(i, j)$
$\lor\ SendVote(i, j)$
$\lor\ RecvVote(i, j)$
$\lor\ BecomeLeader(i, Q)$
$\lor\ Decide(i, v)$

Protocol actions.

$SendRequestVote(src, dst) \triangleq$
$\land\ voteRequestMsg' = voteRequestMsg \cup \{\langle src, dst\rangle\}$

$SendVote(src, dst) \triangleq$
$\land\ \neg voted[src]$
$\land\ \langle dst, src\rangle \in voteRequestMsg$
$\land\ voteMsg' = voteMsg \cup \{\langle src, dst\rangle\}$
$\land\ voted'[src] := \text{True}$
$\land\ voteRequestMsg' = voteRequestMsg \setminus \{\langle src, dst\rangle\}$

$RecvVote(n, sender) \triangleq$
$\land\ \langle sender, n\rangle \in voteMsg$
$\land\ votes'[n] := votes[n] \cup \{sender\}$

$BecomeLeader(n, Q) \triangleq$
$\land\ Q \subseteq votes[n]$
$\land\ leader'[n] := \text{True}$

$Decide(n, v) \triangleq$
$\land\ leader[n]$
$\land\ decided[n] = \{\}$
$\land\ decided'[n] := \{v\}$

Fig. 1. State variables, initial states (*Init*), transition relation (*Next*) for the *SimpleConsensus* protocol. Definitions of the protocol actions are shown on the right.

$NoConflictingValues \triangleq$ Safety property.
$\forall n_1, n_2 \in Node, v_1, v_2 \in Value :$
$(v_1 \in decided[n_1] \land v_2 \in decided[n_2]) \Rightarrow (v_1 = v_2)$

$UniqueLeaders \triangleq$
$\forall n_1, n_2 \in Node :\ leader[n1] \land leader[n_2] \Rightarrow (n_1 = n_2)$

$LeaderHasQuorum \triangleq$
$\forall n \in Node : leader[n] \Rightarrow$
$(\exists Q \in Quorum : votes[n] = Q)$

$LeadersDecide \triangleq$
$\forall n \in Node : (decided[n] \neq \{\}) \Rightarrow leader[n]$

$Ind \triangleq$ Inductive invariant.
$\land\ NoConflictingValues$
$\land\ UniqueLeaders$
$\land\ LeaderHasQuorum$
$\land\ LeadersDecide$
$\land\ NodesVoteOnce$
$\land\ VoteRecordedImpliesVoteMsg$
$\land\ VoteMsgsUnique$
$\land\ VoteMsgImpliesNodeVoted$

Fig. 2. Top-level safety property, *NoConflictingValues*, and complete inductive invariant, *Ind*, for proving its safety in the *SimpleConsensus* from Figure 1. Selected lemma definitions from *Ind* also shown.

difficult to manage and understand the global structure of these inductive invariants as they are being developed, since the interaction between existing lemmas of the invariant candidate and the actions of the system become complex.

For example, consider the development of *Ind*, the complete inductive invariant for *SimpleConsensus* shown in Figure 2. Suppose some partial progress has been made, and the current inductive

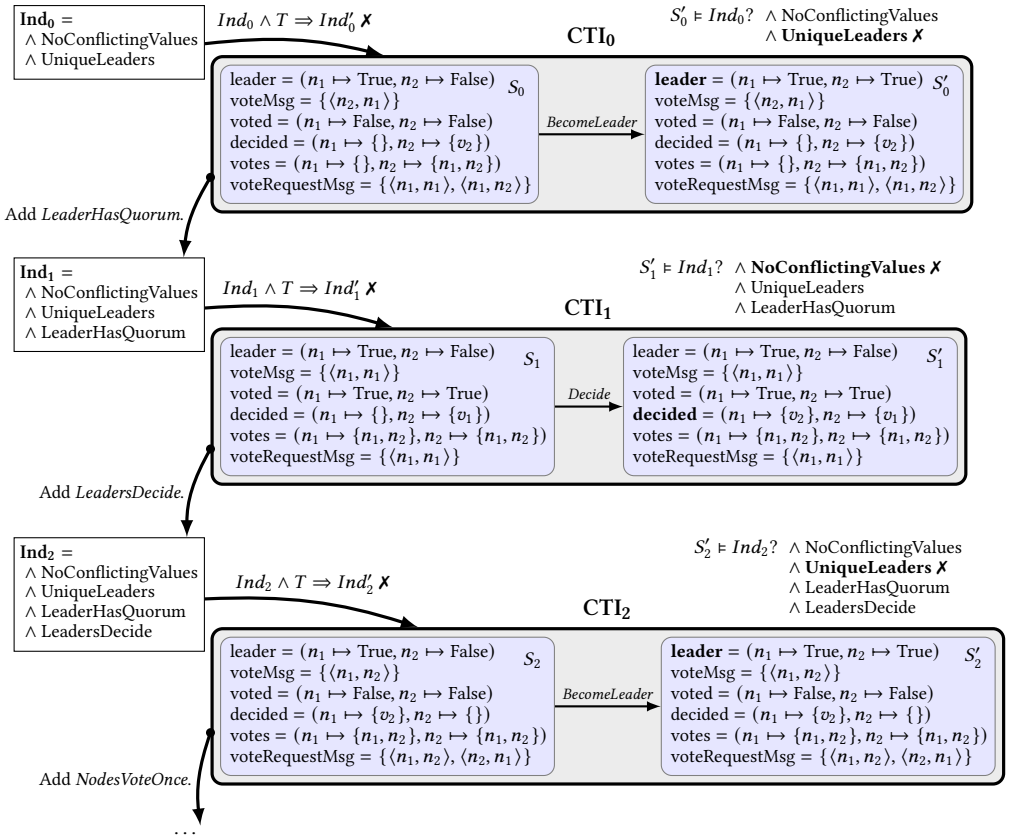


Fig. 3. Possible CTI generation and analysis sequence for a standard inductive invariant development procedure for the *SimpleConsensus* protocol shown in Figure 1. Modified state variables in CTIs are bolded.

invariant candidate consists of the initial 2 lemmas, as follows:

$$\begin{aligned} Ind_0 \triangleq &\wedge NoConflictingValues \\ &\wedge UniqueLeaders \end{aligned} \tag{1}$$

The basic next step in this process is to generate a counterexample to induction (CTI) for Ind_0, analyze this CTI, and develop a lemma invariant that rules it out.

At this point, a user is already faced with several questions about how to proceed effectively. First, there may be many possible CTIs that exist for a current inductive invariant candidate, and one of these CTIs must be generated and selected for analysis. Without an explicit CTI management strategy, it is possible for a user's sequence of CTI analyses to end up context switching between different underlying proof obligations, which can be inefficient and cognitively burdensome. For example, as seen in the sample analysis sequence illustrated in Figure 3, $\mathbf{CTI_0}$ and $\mathbf{CTI_2}$ are, in fact, both possible CTIs for Ind_0, and are both associated with violations of the *UniqueLeaders* lemma. Thus, the corresponding lemmas developed to rule out each of these CTIs (*LeaderHasQuorum* and *NodesVoteOnce*) are needed to ensure that *UniqueLeaders* holds inductively. As shown in Figure 3, however, $\mathbf{CTI_0}$ and $\mathbf{CTI_2}$ are interrupted by the presentation of $\mathbf{CTI_1}$, which is associated with

the violation of an unrelated lemma, *NoConflictingValues*. For large proofs with large numbers of outstanding CTIs, this context switching and management problem is only exacerbated.

In addition to this CTI management and context switching issue, during analysis of a particular CTI it is difficult to determine which aspects of the protocol are actually relevant to eliminating that CTI. For example, if we consider $\mathbf{CTI_1}$ from Figure 3, a basic part of the analysis task is to determine why $\mathbf{CTI_1}$ is unreachable in our system. A precise analysis may observe that *NoConflictingValues* is the lemma invariant violated by this CTI in state S'_1, which helps us focus on the *Decide* action and *NoConflictingValues* lemma pair, narrowing our analysis to the $\{leader, decided\}$ set of state variables. After more thought, a user might discover that this violation is due to the fact that a key property of the system has been violated, namely, the property that only leaders could have decided a value. The user would develop a new lemma invariant, *LeadersDecide*, add it to the set of lemma invariants, and proceed to check the inductiveness of the new, strengthened inductive invariant candidate. This sequence of reasoning represents a typically manual process of localizing one's analysis to a particular subset of relevant protocol state. In existing approaches, though, there exists no systematic technique for performing this kind of localization.

As a proof proceeds, it is also difficult to understand the current status and structure of the overall inductive invariant. For example, after addition of a new lemma, it is not clear whether it introduced new CTIs, or discharged any existing inductive proof obligations of the current invariant candidate, or both. We also have little view into the logical dependencies between lemmas of the current inductive invariant e.g., in the sense of how one lemma depends inductively on another, or a subset of other lemmas, making it difficult to understand the overall proof structure as we proceed.

In summary, we characterize the above issues with existing invariant development approaches into the following broad themes:

Q1. **CTI Management**: How does one manage the set of CTIs for the current inductive invariant and decide which CTI from this set to analyze?
Q2. **Localization**: Once a CTI is selected, how does one focus their analysis only on the lemmas, actions, and state variables that are relevant to analysis of this CTI?
Q3. **Proof Status and Structure**: As new lemmas are developed, how can the current proof status, structure, and progress be measured?

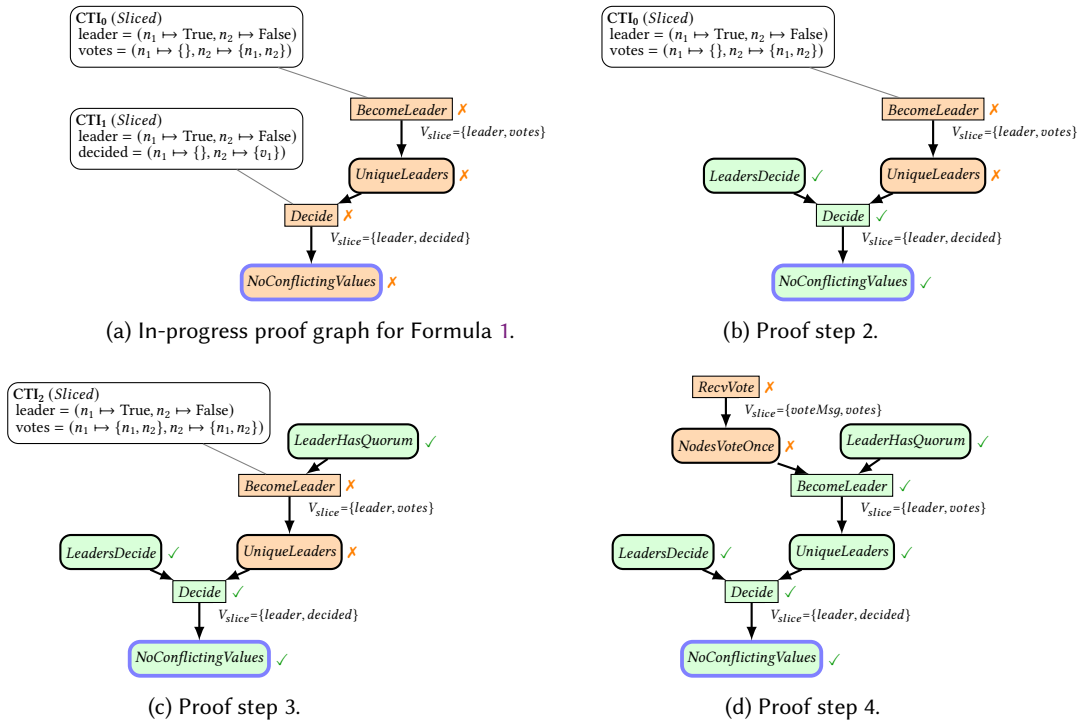
Our work in this paper is largely motivated by the fact that, to our knowledge, no existing proof methodologies provide a formal, conceptual framework for addressing the above questions, which makes the counterexample-guided inductive invariant development process one that can be opaque and extremely laborious even for relatively experienced protocol designers or human verifiers.

2.2 Our Approach: Inductive Proof Decomposition

To address the limitations of existing techniques as described above, our proof methodology, *inductive proof decomposition*, centers around applying a new, fundamental type of decomposition to the structure of an inductive invariant. This decomposition gives rise to the *inductive proof graph*, a novel structure we define and use to guide proof development.

We illustrate the benefits of our approach by working through development of the *SimpleConsensus* partial inductive invariant from Section 2.1, contrasting these to the problems faced in existing approaches.

2.2.1 Our Interactive Verification Procedure. Figure 4a shows an in-progress inductive proof graph that corresponds to the partially completed inductive invariant from Formula 1. The main nodes of this graph, *lemma nodes*, correspond to lemmas of a system (so can be mapped to lemmas of a standard inductive invariant), and the edges represent *relative induction* dependencies between



(a) In-progress proof graph for Formula 1.

(b) Proof step 2.

(c) Proof step 3.

(d) Proof step 4.

Fig. 4. Example progression of inductive proof graph development. Nodes in orange with ✗ are those with remaining inductive proof obligations to be discharged, and those in green with ✓ represent those with all obligations discharged. Note that only pre-states of CTIs are shown, for brevity, and that CTIs are not a part of the proof graph itself but shown as annotations associated with action nodes.

these lemmas. This dependency structure is also decomposed by protocol actions, represented in the graph via *action nodes*, which are associated with each lemma node, and map to distinct protocol actions (e.g., the actions of *SimpleConsensus* listed in Figure 1). Formally, an action and lemma node pair (L, A) is associated with a corresponding proof obligation $Supp_A \wedge L \wedge A \Rightarrow L'$, where $Supp_A$ is the conjunction of incoming lemma nodes to action node A.

Our invariant development process now starts from the partial inductive proof graph shown in Figure 4a, from which the possible next steps in our process are now significantly easier to assess. In particular, it is clear that the *Decide* and *BecomeLeader* nodes are unproven (shown in orange and marked with ✗), meaning that there are outstanding CTIs for the inductive proof obligations of those nodes as explained above. Additionally, we can focus on separate CTIs in isolation, since CTIs are now associated with specific action nodes. This makes it clear which lemmas and actions the counterexamples are relevant to, alleviating the issues raised in Q1 discussed above.

To proceed in the development of our inductive invariant, we extend this graph by developing appropriate support lemmas and associated edges. For example, we may first select $\mathbf{CTI_1}$ to analyze, as shown in Figure 4a. In addition to the localization of counterexamples, the decomposition provided by the proof graph also allows for localized state variable projection to be applied to the CTIs at each action node. These projected variable sets, which we refer to as *variable slices*, are shown as V_{slice} alongside each action node. For example, at the *Decide* node in Figure 4a, which $\mathbf{CTI_1}$ is associated with, $V_{slice} = \{leader, decided\}$, containing only 2 out of 6 total state variables

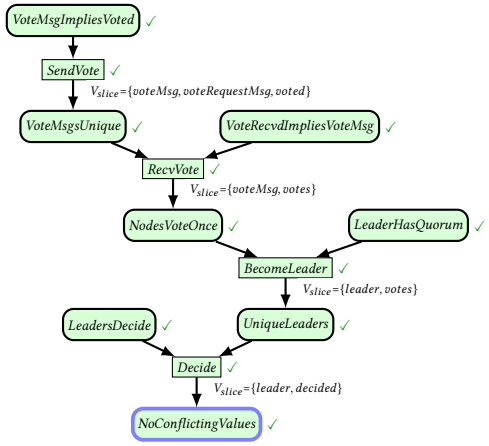


Fig. 5. Complete inductive proof graph for *SimpleConsensus* and safety property *NoConflictingValues*. Lemma nodes are depicted as rounded boxes and action nodes as rectangles.

of the *SimpleConsensus* system. These slices can be computed by a particular static analysis of an action-lemma pair and, as seen in this example, this technique often significantly reduces the number of state variables to be considered at each node. This can greatly reduce the analysis burden on a human user, addressing the issues raised in Q2 above.

After analyzing $\mathbf{CTI_1}$ we may develop the *LeadersDecide* lemma, which states that only leaders could have decided values, and rules out $\mathbf{CTI_1}$. *LeadersDecide* is then added as a new incoming support lemma of the *Decide* action node, leading us to the proof graph state shown in Figure 4b. At this point, we see from the graph status that lemmas *NoConflictingValues* and *LeadersDecide* are discharged, so we no longer need to consider these lemmas in our reasoning. This gives us a useful dynamic measure of proof status and logical structure as we develop new lemmas over time, touching upon the issues discussed in Q3 above.

Next, as shown in Figure 4b, we have a remaining counterexample, $\mathbf{CTI_0}$, associated with the unproven *BecomeLeader* node. Here, we again benefit from local counterexample slicing, which gives $V_{slice} = \{leader, votes\}$ at this *BecomeLeader* node. Analysis of $\mathbf{CTI_0}$ yields the *LeaderHasQuorum* support lemma, which is self-inductive, leading us to the proof state in Figure 4c. From there, we develop one additional support lemma to rule out $\mathbf{CTI_2}$, giving us the *NodesVoteOnce* support lemma, which is sufficient to discharge the *UniqueLeaders* lemma node, leading us to the proof state shown in Figure 4d. We do not show a further progression of the proof process for this graph, but we can see from Figure 4d that this process can be continued in a backwards fashion, starting now from the remaining unproven node *NodesVoteOnce* and its associated action node $RecvVote$. A fully completed proof graph is shown in Figure 5, where every node has been discharged i.e., has a valid set of support lemmas.

Although this is a relatively small protocol and invariant, this example begins to demonstrate the value of our interactive, compositional proof methodology. Our decomposition based on the inductive proof graph structure makes explicit the relationship between lemmas and protocol actions as the inductive invariant is being built, and, enabled by this structure, we are able to localize the analysis of CTIs to specific nodes of this graph. Additionally, this counterexample localization enables our *slicing* technique, which projects out irrelevant state from the CTIs that need to be analyzed by a human verifier.

In the remainder of this paper, we formalize the above ideas in detail, and present a more extensive evaluation applying this proof methodology to several complex protocols.

3 PRELIMINARIES

We are focused on the problem of safety verification of protocols formalized as discrete transition systems, which consists of a core problem of finding adequate inductive invariants. Furthermore, we are focused on verification of systems that are assumed to be correct i.e., we assume various bug-finding methods ([17],[4]) have been applied upfront before a proof is undertaken.

Transition Systems and Invariants. The protocols considered in this paper can be modeled as *symbolic transition systems*, where a state predicate I defines the possible values of state variables at initial states of the system, and a predicate T defines the *transition relation*. A transition system M is then defined as $M = (I, T)$, and the *behaviors* of M are defined as the set of all sequences of states $\sigma_1 \rightarrow \sigma_2 \rightarrow \dots$ that begin in some state satisfying I and where every transition $\sigma_i \rightarrow \sigma_{i+1}$ satisfies T. The *reachable states* of M are the set of all states that exist in some behavior. In this paper we are concerned with the verification of *invariants*, which are predicates over the state variables of a system that hold true at every reachable state of a system M. In this paper, we also assume that the transition relation T for a system M is composed of distinct logical actions, $T = A_1 \vee \dots \vee A_k$. For example, a simple transition relation of this form is $T = (x' = x + 1) \vee (x' = x + 2)$, where a primed state variable (x') represents the value of that state variable in the next state.

Guarded Actions. We also define a restricted class of systems where transition relations are expressed in a *guarded action* style. That is, systems where all actions A are of the form $A = Pre \wedge Post$, where Pre is a predicate over current state variables and $Post$ is a conjunction of update formulas of the form $x_i' = f_i(\mathcal{D}_i)$, where f_i is some expression over a subset of current state variables $\mathcal{D}_i$. For simplicity, we assume that all state variables always appear in $Post$, and that for variables unchanged by a protocol action, they simply appear in $Post$ with an identity update expression, $x_i' = x_i$. Note that although systems in guarded action style have deterministic update expressions, these systems can still be non-deterministic, due to non-determinism over constant system parameters, e.g., as seen in *SimpleConsensus* in Figure 1.

Inductive Invariants and Relative Induction. The standard technique for proving an invariant S of a system $M = (I, T)$ is to develop an *inductive invariant* [34], which is a state predicate Ind such that $Ind \Rightarrow S$ and

$$I \Rightarrow Ind \tag{2}$$

$$Ind \wedge T \Rightarrow Ind' \tag{3}$$

where Ind' denotes the predicate Ind where state variables are replaced by their primed, next-state versions. Conditions (2) and (3) are, respectively, referred to as *initiation* and *consecution*. Condition (2) states that Ind holds at all initial states. Condition (3) states that Ind is *inductive*, i.e., if it holds at some state s then it also holds at any successor of s. Together these two conditions imply that Ind is also an invariant, i.e., that Ind holds at all reachable states.

Typically, an inductive invariant is represented as a strengthening of S via a conjunction of smaller *lemma invariants*, $L_1, \dots, L_k$, such that the final inductive invariant is defined as $Ind = S \wedge L_1 \wedge \dots \wedge L_k$. Throughout this paper we assume inductive invariants can be represented in this form. Note also that for a given system $M = (I, T)$, a state predicate may be inductive only under the assumption of some other predicate. For given state predicates Ind and L, if the formula $L \wedge Ind \wedge T \Rightarrow Ind'$ is valid, we say that Ind is *inductive relative to* L.

4 INDUCTIVE PROOF GRAPHS

Our approach to interactive inductive invariant development is based on a core logical structure, the *inductive proof graph*, which we discuss and formalize in this section. This graph encodes the structure of an inductive invariant in a way that is amenable to localized reasoning and human interpretability, and can naturally be decomposed into discrete proof obligations. We begin at a high level in Section 4.1 by discussing a basic version of this graph based on relative induction dependencies. In Section 4.2 we give the full definition of the structure, which takes advantage of action-based decomposition. Section 5 then describes our proof approach based on this structure, *inductive proof decomposition*, where a human verifier incrementally constructs this structure by working backwards from a target safety property.

4.1 Relative Induction Graph

A *monolithic* approach to inductive invariant development, where one searches for a single inductive invariant that is a conjunction of smaller lemmas, is a general proof methodology for safety verification [34]. Any monolithic inductive invariant, however, can alternatively be viewed in terms of its *relative induction* dependency structure, which is the initial basis for our formalization of the inductive proof graph structure.

Namely, for a transition system $M = (I, T)$ and associated invariant S, given an inductive invariant

$$Ind = S \wedge L_1 \wedge \cdots \wedge L_k$$

each lemma in this overall invariant may only depend inductively on some other subset of lemmas in Ind. More formally, proving the consecution step of such an invariant requires establishing validity of the following formula

$$(S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T \Rightarrow (S \wedge L_1 \wedge \cdots \wedge L_k)' \tag{4}$$

which can be decomposed into the following set of independent proof obligations:

$$\begin{aligned} (S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T &\Rightarrow S' \\ (S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T &\Rightarrow L_1' \\ &\vdots \\ (S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T &\Rightarrow L_k' \end{aligned} \tag{5}$$

If the overall invariant Ind is inductive, then each of the proof obligations in Formula 5 must be valid. That is, each lemma in Ind is inductive relative to the conjunction of all other lemmas in Ind.

With this in mind, if we define $\mathcal{L} = \{S, L_1, \ldots, L_k\}$ as the lemma set of Ind, we can consider the notion of a *support set* for a lemma in $\mathcal{L}$ as any subset $U \subseteq \mathcal{L}$ such that L is inductive relative to the conjunction of lemmas in U i.e., $(\bigwedge_{\ell \in U} \ell) \wedge L \wedge T \Rightarrow L'$. As shown above in Formula 5, $\mathcal{L}$ is always a support set for any lemma in $\mathcal{L}$, but it may not be the smallest support set. This support set notion gives rise a structure we refer to as the *lemma support graph*, which is induced by each lemma's mapping to a given support set, each of which may be much smaller than $\mathcal{L}$. Figure 6 shows an example of an abstract lemma support graph along with the corresponding proof obligations that map to each node of the graph.

4.2 Action Decomposition

The initial definition of lemma support graphs given above considers lemmas as nodes of this graph, and support edges as running from one lemma to another. For distributed and concurrent protocols, however, the transition relation of a system $M = (I, T)$ is typically a disjunction of several distinct actions i.e., $T = A_1 \vee \cdots \vee A_n$, as described in Section 3. This provides an additional layer

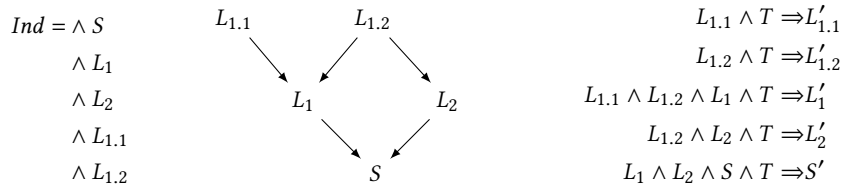


Fig. 6. A lemma support graph for an inductive invariant *Ind*, and corresponding proof obligations.

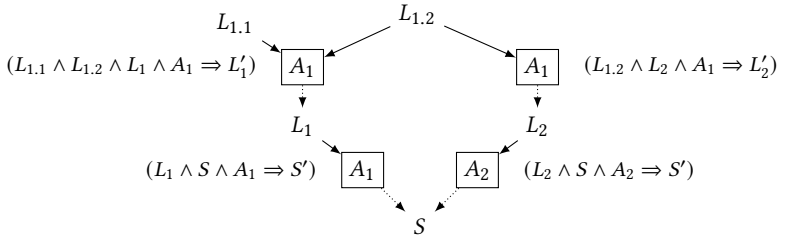


Fig. 7. Inductive proof graph based on Figure 6, with action decomposition applied. Action nodes are depicted as boxes, and associated inductive proof obligations are shown in parentheses next to each action node. Self-inductive obligations are omitted for brevity. Action to lemma node relationships are shown as incoming dotted edges.

of decomposition that can be applied to the lemma support graph notion, which gives rise to the full definition of the inductive proof graph structure.

Each node of a lemma support graph is augmented with sub-nodes, one for each action of the overall transition relation. Lemma support edges in the graph then run from a lemma to a specific action node, rather than directly to a target lemma. Incorporation of this action-based decomposition now lets us define the full inductive proof graph structure.

Definition 4.1. For a system $M = (I, T)$ with $T = A_1 \vee \cdots \vee A_k$, an *inductive proof graph* is a directed graph (V, E) where

- $V = V_L \cup V_A$ consists of a set of *lemma nodes* V_L and *action nodes* V_A, where
  - V_L is a set of state predicates over M.
  - $V_A = V_L \times \{A_1, \ldots, A_k\}$ is a set of action nodes, associated with each lemma node in V_L.
- $E \subseteq V_L \times V_A$ is a set of *lemma support edges*.

Figure 7 shows an example of an inductive proof graph along with its corresponding inductive proof obligations annotating each action node. Note that, for simplicity, when depicting inductive proof graphs, if an action node is self-inductive, we omit it. Also, action nodes are, by default, always associated with a particular lemma, so when depicting these graphs, we show dotted edges that connect action nodes to their parent lemma node, even though these edges do not appear in the formal definition.

4.2.1 Proof Graph Validity. We now define a notion of *validity* for an inductive proof graph. That is, we define conditions on when a proof graph can be seen as corresponding to a complete inductive invariant and, correspondingly, when the lemmas of the graph can be determined to be invariants of the underlying system.

Definition 4.2 (Local Action Validity). For an inductive proof graph $(V_L \cup V_A, E)$, let the *inductive support set* of an action node $(L, A) \in V_A$ be defined as $Supp_{(L,A)} = \{\ell \in V_L : (\ell, (L, A)) \in E\}$. We then say that an action node (L, A) is *locally valid* if the following holds:

$$\left(\bigwedge_{\ell \in Supp_{(L,A)}} \ell\right) \wedge L \wedge A \Rightarrow L' \tag{6}$$

Definition 4.3 (Local Lemma Validity). For an inductive proof graph $(V_L \cup V_A, E)$, a lemma node $L \in V_L$ is *locally valid* if all of its associated action nodes, $\{L\} \times \{A_1, \ldots, A_k\}$, are locally valid.

Based on the above local validity definitions, the notion of validity for a full inductive proof graph is then straightforward to define.

Definition 4.4 (Inductive Proof Graph Validity). An inductive proof graph is *valid* whenever all lemma nodes of the graph are *locally valid*.

The validity notion for an inductive proof graph establishes lemmas of such a graph as invariants of the underlying system M, since a valid inductive proof graph can be seen to correspond with a complete inductive invariant. We formalize this as follows.

Lemma 4.5. *For a system $M = (I, T)$, if an inductive proof graph $(V_L \cup V_A, E)$ for M is valid, and $I \Rightarrow L$ for every $L \in V_L$, then the conjunction of all lemmas in V_L is an inductive invariant.*

Proof. The conjunction of all lemmas in a valid graph must be an inductive invariant, since every lemma's support set exists as a subset of all lemmas in the proof graph, and all lemmas hold on the initial states. □

Theorem 4.6. *For a system $M = (I, T)$, if a corresponding inductive proof graph $(V_L \cup V_A, E)$ for M is valid, and $I \Rightarrow L$ for every $L \in V_L$, then every $L \in V_L$ is an invariant of M.*

Proof. By Lemma 4.5, the conjunction of all lemmas in a valid proof graph is an inductive invariant, and for any set of predicates, if their conjunction is an invariant of M, then each conjunct must be an invariant of M. □

Note on Cycles and Subgraphs. Note that the definition of proof graph validity does not imply any restriction on cycles in a valid inductive proof graph. For example, a proof graph that is a pure k-cycle can be valid. For example, consider a simple *ring counter* system with 3 state variables, a, b, and c, where a single value gets passed from a to b to c and exactly one variable holds the value at any time. An inductive invariant establishing the property that a always has a well-formed value will consist of 3 properties that form a 3-cycle, each stating that a,b and c's state are, respectively, always well-formed.

Also note that based on the above validity definition, any subgraph of an inductive proof graph can also be considered valid, if it meets the necessary conditions. Thus, in combination with Theorem 4.6 this implies that, even if a particular proof graph is not valid, there may be subgraphs that are valid and, therefore, can be used to infer that a subset of lemmas in the overall graph are valid invariants.

5 INTERACTIVE SAFETY VERIFICATION

Having formalized the inductive proof graph structure, we now describe our proof methodology, *inductive proof decomposition*, which is based around incremental construction of an inductive proof graph, with integration of localized counterexample guidance.

Procedure 1 High level steps of our interactive inductive invariant development procedure.

1: **Inputs**: Transition system M, invariant S.
2: **Initialization**:
3: $V_L \leftarrow \{S\}$
4: $V_A \leftarrow \{S\} \times \{A_1, \ldots, A_k\}$
5: $E \leftarrow \emptyset$
6: $G \leftarrow (V_L \cup V_A, E)$
7: **procedure** IndProofDecomp
8: **if** all lemmas V_L of G are locally valid **then**
9: **return** G, a valid inductive proof graph.
10: **else**
11: Choose some $(L, A) \in V_A$ such that (L, A) is not locally valid.
12: Analyze a CTI X of node (L, A), develop a lemma L_{new} that eliminates X and update G as:
13: $V_L \leftarrow V_L \cup \{L_{new}\}$
14: $V_A \leftarrow V_A \cup (\{L_{new}\} \times \{A_1, \ldots, A_k\})$
15: $E \leftarrow E \cup \{(L_{new}, (L, A))\}$
16: **goto** Line 8.
17: **end if**
18: **end procedure**

5.1 Interactive Verification Procedure

Our high level interactive safety verification procedure, for proving that S is an invariant of transition system $M = (I, T)$ by finding an inductive invariant Ind, centers around the incremental construction of an inductive proof graph, working backwards from the target safety property, S.

That is, we begin with an inductive proof graph $(V_L \cup V_A, E)$ where $V_L = \{S\}, V_A = \{S\} \times \{A_1, \ldots, A_k\}$, and $E = \emptyset$. From here, the graph is extended incrementally by developing support lemmas and adding support edges from these lemmas to action nodes that are not yet locally valid in the current graph. This is described more precisely in Procedure 1. Note there that, in line 12, L_{new} may be an existing lemma in the graph, in which case only E will be modified in the subsequent steps. Note also that, as mentioned in Section 2, we assume access to bounded verification tools for checking invariants that are added as new lemmas to an inductive proof graph (e.g., BMC [4]). A sample progression of this procedure for a concrete protocol is illustrated in Figure 4, as explained previously in Section 2.2.

In the following section we describe key features of this procedure that are enabled by the inductive proof graph structure, centered around localized counterexample guidance and our *slicing* technique that can be applied at each action node of a proof graph.

5.2 Localized Counterexample Guidance

Managing and analyzing counterexamples to induction (CTIs) is a key aspect of an inductive safety verification methodology that relies on machine assistance in the form of induction checks. Our approach takes advantage on the compositional structure of an inductive proof graph to scope induction counterexamples to action nodes of the proof graph. Specifically, if an action node (L, A) is not locally valid in the current proof graph, then we associate with that node CTIs which are violations of the associated proof obligation shown in Formula 6.

This provides a user the ability to consider counterexamples in the particular scope of an action and lemma pair, without considering them in the global scope of all lemmas developed so far, as illustrated previously in the example shown in Figure 4. In addition, this counterexample localization enables a projection-based abstraction of presented counterexamples that we refer to as *counterexample slicing*, which we formalize in the following section.

Counterexample Slicing. When considering an action node (L, A), any support lemmas for this node must, to a first approximation, refer only to state variables that appear in either L or A. We can make use of this general idea to compute a *variable slice* for counterexamples presented at each node. That is, we remove from consideration any state variables that are irrelevant for establishing a valid support set for that node. Intuitively, the variable slice of an action node (L, A) can be understood as the union of: (1) the set of all variables appearing in the precondition of A, (2) the set of all variables appearing in the definition of lemma L, (3) for any variables in L, the set of all variables upon which the update expressions of those variables depend.

More precisely, our slicing computation at each action node is based on the following static analysis of a lemma and action pair (L, A). First, let $\mathcal{V}$ be the set of all state variables in our system, and let $\mathcal{V}'$ refer to the primed, next-state copy of these variables. For an action node (L, A), we have $L \wedge A \Rightarrow L'$ as its initial inductive proof obligation. As noted in Section 3, we consider actions to be written in *guarded action* form, so they can be expressed as $A = Pre \wedge Post$, where Pre is a predicate over a set of current state variables, denoted $Vars(Pre) \subseteq \mathcal{V}$, and $Post$ is a conjunction of update expressions of the form $x_i' = f_i(\mathcal{D}_i)$, where $x_i' \in \mathcal{V}'$ and $f_i(\mathcal{D}_i)$ is an expression over a subset of current state variables $\mathcal{D}_i \subseteq \mathcal{V}$.

Definition 5.1. For an action $A = Pre \wedge Post$ and variable $x_i' \in \mathcal{V}'$ with update expression $f_i(\mathcal{D}_i)$ in $Post$, we define the *cone of influence* of x_i', denoted $COI(x_i')$, as the variable set $\mathcal{D}_i$. For a set of primed state variables $\mathcal{X} = \{x_1', \ldots, x_k'\}$, we define $COI(\mathcal{X})$ simply as $COI(x_1') \cup \cdots \cup COI(x_k')$

Now, if we let $Vars(Pre) \subseteq \mathcal{V}$ and $Vars(L') \subseteq \mathcal{V}'$ be the sets of state variables that appear in the expressions of L' and Pre, respectively, then we can formally define the notion of a slice as follows.

Definition 5.2. For an action node (L, A), its *variable slice* is the set of state variables

$$Slice(L, A) = Vars(Pre) \cup Vars(L) \cup COI(Vars(L'))$$

Based on this definition, we can now show that a variable slice is a strictly sufficient set of variables to consider when developing a support set for an action node.

Theorem 5.3. *For an action node (L, A), if a valid support set exists, there must exist one whose expressions refer only to variables in $Slice(L, A)$.*

Proof. Without loss of generality, the existence of a support set for (L, A) can be defined as the existence of a predicate $Supp$ such that the formula

$$Supp \wedge L \wedge A \wedge \neg L' \tag{7}$$

is unsatisfiable. As above, actions are of the form $A = Pre \wedge Post$, where $Post$ is a conjunction of update expressions, $x_i' = f_i(\mathcal{D}_i)$, so Formula 7 can be re-written as

$$Supp \wedge L \wedge Pre \wedge \neg L'[Post] \tag{8}$$

where $L'[Post]$ represents the expression L' with every $x_i' \in Vars(L')$ substituted with the update expression given by $f_i(\mathcal{D}_i)$. From this, it is straightforward to show our original goal. If $L \wedge Pre \wedge \neg L'[Post]$ is satisfiable, and there exists a $Supp$ that makes Formula 8 unsatisfiable, then clearly $Supp$ must only refer to variables that appear in $L \wedge Pre \wedge \neg L'[Post]$, which are exactly the set of variables in $Slice(L, A)$.

□

6 EMPIRICAL EVALUATION

To evaluate our technique, we developed inductive invariants for several distributed protocols of varying complexity, including development of inductive invariants for two large scale protocol specifications of the Raft [38] and Zab [23] distributed consensus protocols. Overall, the core goals of our evaluation were to (1) understand the empirical structure of inductive proof graphs for real world protocols, (2) evaluate the effectiveness of our proof technique for developing inductive invariants of large protocols, and (3) examine how these proof graphs provide insight into understanding the intuitive structure of an inductive proof.

We describe our experience developing proofs for these protocols using our technique and provide metrics on the compositional structure of these protocols and their invariants. Note that all code for our implementations and protocol benchmarks described below is available in the supplementary material for this paper, along with instructions for running the tool and viewing and checking our proofs.

Implementation and Setup. We implemented our technique in a tool, Indigo, that provides a graphical user interface for the interactive development of inductive proof graphs. Indigo is implemented in Python and accepts systems specified in the TLA+ specification language [30]. The tool allows a user to view a current inductive proof graph, with visual coloring representing the state of each lemma and action node. A user can then choose to focus on a particular action node, and a subset of CTIs associated with that node are presented. Slicing is applied automatically to the presented CTIs based on a static analysis of the protocol specification, and the interface also provides a way to for a user to dynamically add support edges and new lemmas to the graph.

Internally, Indigo tool uses a combination of the TLC model checker [52] and the Apalache symbolic model checker [27] for checking inductive proof obligations and generating counterexamples to induction. Note that TLC is an explicit state model checker, so cannot produce proofs of inductive invariants for protocols of nontrivial size, but can generate counterexamples to induction for finite protocol instances using a randomized search technique [31]. Apalache generates an SMT [1] encoding of a TLA+ specification which is passed internally to the Z3 solver [9]. Apalache can generate CTIs and also produce proofs of inductive invariants for bounded protocol parameters. In our experience, TLC can, in some cases, be more efficient at generating CTIs than Apalache, so we found both tools useful in our implementation. All experiments and proof checking results below were collected using Apalache version 0.44.0 and TLC version 2.15 running on a 2020 M1 Macbook Air.

Protocol Benchmarks. We used our tool to develop inductive invariants for establishing core safety properties of 5 distributed protocol specifications. These protocols are summarized in Table 1, along with various specification and proof statistics. All formal specifications of these protocols are defined in TLA+, some of which existed from prior work and some of which we developed or modified based on existing specifications. All protocols are parameterized (e.g., in the number of nodes/servers, epochs, etc.), and we completed our inductive invariant development efforts for fixed parameter bounds, which allows the inductive proof obligations to be checked by Apalache. We chose to use fixed parameter bounds to make proof obligation checking more feasible, and since we were focused mainly on exploring the inductive invariant development process and not on the process of proving a developed inductive invariant.

Of the protocols tested, we consider the following 3 to be of medium complexity:

- *SimpleConsensus*: An abstract consensus protocol where nodes vote to elect a leader which then decides on a value, as discussed previously in Section 2. Parameters used include the set of nodes, $Node = \{n_1, n_2, n_3, n_4\}$, and the set of values, $Value = \{v_1, v_2, v_3, v_4\}$.

Protocol	Variables	Actions	LOC (spec/pf)	Lemmas	Median slice/indeg	Check (s)
SimpleConsensus	6	5	77/ 51	8	2 (0.33) / 1	72
TwoPhase	5	7	179/ 129	16	3 (0.60) / 1	184
AbstractRaft	4	6	220/ 138	10	3 (0.75) / 1	969
AsyncRaft	12	15	608/ 553	39	5 (0.42) / 2	6592
Zab	23	16	1221/ 774	33	9 (0.39) / 1	1101

Table 1. Protocols used in evaluation and metrics on their specifications and inductive proof graphs. *LOC* is the number of lines of TLA+ code for defining the specification (spec) and inductive proof (pf), respectively. *Median slice / indeg* gives the median size of all variable slices in the complete inductive proof graph, and its proportion of the total state variables / the median number of total incoming edges to action nodes. *Check (s)* is the time to check the completed proof using Apalache, in seconds. The current Zab proof graph was checked using TLC.

- *TwoPhase*: A specification of the two-phase commit protocol [32], where a transaction manager coordinates a set of resource managers to consistently agree on a *commit* or *abort* decision. The safety property checked is *TCConsistent*, which states that two resource managers cannot have conflicting commit or abort decisions. Parameters used include the set of resource managers $RM = \{s_1, s_2, s_3, s_4\}$.
- *AbstractRaft*: An abstract specification of the Raft consensus protocol [38], which abstracts away low level message passing details. Safety property is *StateMachineSafety*, which states that log entries committed at the same indices must be the same. Parameters used include the number of servers, $Server = \{s_1, s_2, s_3, s_4\}$, the maximum length of logs, $MaxLen = 3$, and the maximum number of terms, $MaxTerm = 3$.

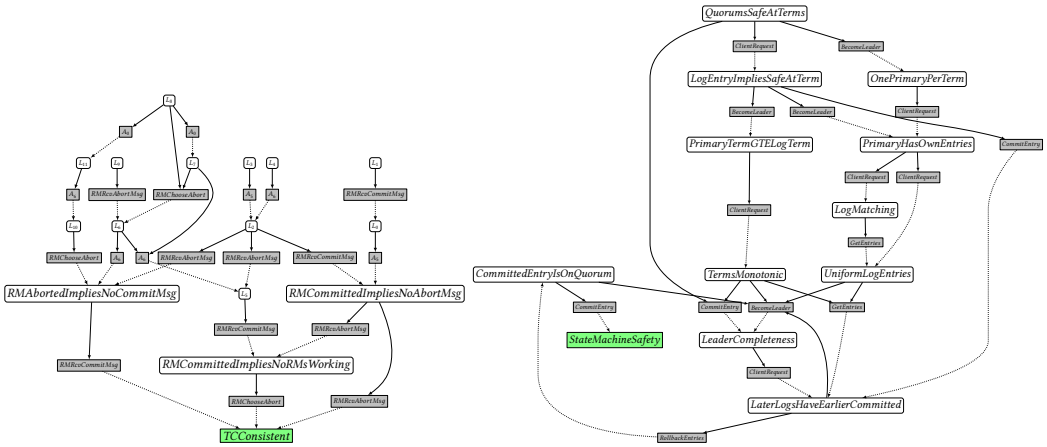
The 2 additional protocols are of significantly larger complexity:

- *AsyncRaft*: An industrial scale specification of the Raft consensus protocol [38]. Our specification is based on [37, 46], and models asynchronous message passing between all nodes and fine-grained local state. The safety property checked is *NoLogDivergence*, which states that if a commit index covers an entry on different nodes, the entries must be consistent. Parameters used include the number of servers, $Server = \{s_1, s_2, s_3\}$, the maximum length of logs, $MaxLen = 3$, and the maximum number of terms, $MaxTerm = 3$.
- *Zab*: An industrial scale specification of the Zab replication protocol [23], which is a totally ordered atomic broadcast protocol for implementing primary backup replication used in Apache Zookeeper [19]. The protocol specification used is based on prior formalization efforts that have been merged into the official Zookeeper code repository [39]. The safety property checked is *PrefixConsistency*, which states that if entries are committed at the same index across nodes, then the entries must be the same. Parameters used include the number of servers, $Server = \{s_1, s_2, s_3\}$, the maximum length of logs, $MaxHistLen = 1$, and the maximum number of epochs $MaxEpoch = 2$.

The 2 large protocol specifications are both of a complexity significantly greater than those tested in recent automated invariant inference techniques, so we consider them as the most relevant benchmarks for evaluating interactive inductive invariant development techniques.

6.1 Results and Discussion

Table 1 shows various statistics about the protocols we tested, including the number of state variables, number of actions, lines of code (LOC) in the TLA+ protocol specifications and proofs, number of lemmas in each proof graph, and the size of variable slices. We discuss the structure of



(a) *TwoPhase* inductive proof graph. (b) *AbstractRaft* inductive proof graph.

Fig. 8. Inductive proof graphs for medium size protocols. Top level safety properties shown in green.

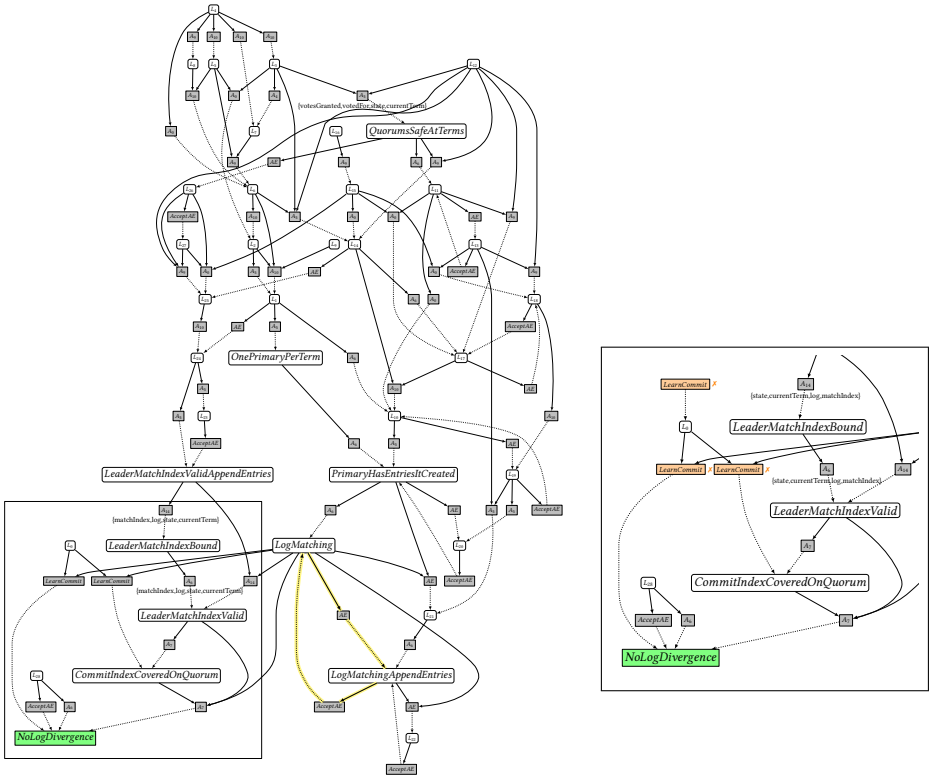
the developed proof graphs and qualitative aspects of our experience developing these proofs in more detail below.

Medium Protocols. The completed inductive proof graphs for *AbstractRaft* and *TwoPhase*, respectively, can be seen in Figures 8a and 8b, and the proof graph for *SimpleConsensus* was shown previously, in Figure 5. The full graphs can be viewed using our interactive tool provided in our supplementary material.

In general, though these proof graphs are for smaller protocols, they generally confirm our intuition that these proof graphs are useful for understanding the structure of an inductive proof and guiding its development. Even for protocols of this size, we found that maintenance of this structure coupled with automatic counterexample slicing is a significant aid, and allowed us to develop these inductive invariants with much more ease and efficiency. In *SimpleConsensus*, for example, the median slice proportion of 0.33, indicating that most slices present a significantly reduced amount of information, and the graph admits a clear tree-like decomposition. Our experience for *TwoPhase* was similar, and also benefits from slicing, with a median slice proportion of 0.6.

The graph for *AbstractRaft* similarly admits a relatively tree-like decomposition, and also provides insight on how lemmas of the protocol relate to each other. For example, we can observe an *induction cycle* related to establishment of key properties about committed log entries. This cycle flows from *CommittedEntryIsOnQuorum* to *LeaderCompleteness* via action *BecomeLeader*, then to *LaterLogsHaveEarlierCommitted* via action *ClientRequest*, then back to *CommittedEntryIsOnQuorum* via *RollbackEntries*. Generally, during development of these inductive proofs, we found this structure helpful to guide our reasoning as it makes the current logical structure of the developed proof explicit.

Large Protocols. As an exploration of our technique for larger scale proof efforts, we applied it first to *AsyncRaft*, a specification of Raft at a considerably lower level of abstraction e.g., it includes asynchronous message passing and fine-grained local state, akin to the detail level provided in the original Raft TLA+ specification [37]. We are also aware of no prior inductive invariant that existed for a TLA+ specification of Raft at this level of complexity and abstraction level. Figure 9 shows our completed inductive proof graph for *AsyncRaft*. Our experience developing this proof



(a) *AsyncRaft* inductive proof graph. (b) Subgraph broken by protocol change.

Fig. 9. *AsyncRaft* inductive proof graph, and excerpt of affected subgraph after breaking protocol change. Variable slices are shown at a subset of nodes, and induction cycle is highlighted in yellow. *AE* action nodes represent an *AppendEntries* action (a leader sends entries), analogously for *AcceptAE*. *LearnCommit* actions represent servers learning of a new commit index.

graph provided several insights into understanding the effectiveness of our technique, and also in understanding the structure of such a proof graph for a large distributed protocol.

Overall, we found our technique effective at making the proof process efficient and understandable for a human verifier, by providing a formal structure to the large scale proof and accelerating CTI analysis by slicing and localized reasoning. We were able to develop such an inductive invariant in approximately 3 human weeks of effort, which we found to be a productive pace for an invariant of this size and protocol of this complexity. Other verification efforts of this type note, for example, an inductive invariant development burden of 1-2 human months [42], even for a protocol with a smaller invariant than *AsyncRaft*. We found that the decomposition provided by our technique enabled effective local reasoning in many cases. For example, comparing the slices near *LeaderMatchIndexValid* and *LeaderMatchIndexBound* nodes to *QuorumsSafeAtTerms*, in a relatively distinct sub-component of the graph, we can see that the slice sets refer to relatively disjoint subsets of variables, supporting our hypothesis that the compositional structure of the proof graph allows for localized reasoning on different aspects of the protocol.

In addition to the efficiency of invariant development, we also found value in the developed proof artifact beyond establishing correctness. For example, during the development of the proof,

we found it helpful to observe various patterns that arise in the proof graph structure. In particular, we observed the various types of *induction cycles* that arise in this graph, especially those that arise in many places due to the message passing nature of this protocol. For example, we can observe the cycle highlighted in Figure 9 where *LogMatching* serves as a support lemma of *LogMatchingInAppendEntriesMsgs* via the *AppendEntries* action, and *LogMatchingInAppendEntriesMsgs* then serves as a support lemma of *LogMatching* via the *AppendEntriesResponse* action, forming this induction cycle. These cycles would be difficult to observe in a standard inductive invariant, but are apparent in our proof graph structure, and represent a common pattern where invariants about protocol state must hold on both local state and also on the state of messages that are sent over the network. Logical dependencies between lemmas also become much clearer in our graph structure. For example, one can observe the position of the *OnePrimaryPerTerm* lemma, which is a core lemma of Raft stating that there must be a unique leader per term, and its ancestral relationship to lemmas like *PrimaryHasEntriesItCreated* and *LogMatching*.

We also found it valuable to examine how changes to a protocol affect such a graph, and how these changes may be used to help evolve a protocol or proof. Figure 9b shows our *AsyncRaft* proof graph after we introduced a breaking change to the specification, modifying a key precondition on when servers are able to learn of a new commit index information. This breaks the top-level safety property *NoLogDivergence*, but we can observe that this fault manifests in a localized manner in the proof graph, with respect to the broken *LearnCommit* action nodes that are colored orange in the graph. A large subgraph of the original graph remains valid, and so, as discussed in Section 4, a large percentage of lemmas in the graph can be seen to remain as invariants of the modified protocol. We view this as a potential means to make repair and evolution of a protocol and its proof a more tractable task.

The inductive proof graph for our other large-scale protocol evaluation, Zab, is shown in Figure 10. We do not give a full analysis here, but we show the graph to give a high level sense of its structure. We note that due to encoding inefficiencies with Apalache for our specification, we verified the current developed inductive proof graph probabilistically using TLC, for the parameters listed above. In general, we found the Zab proof structure somewhat more difficult to develop e.g., there were a greater number of long range dependencies between lemmas, and many lemmas required support via larger sets of actions. It is not clear whether this is due to the underlying protocol itself or the particular specification or lemmas developed.

7 RELATED WORK

Interactive and Compositional Verification. There are two recent works that are most similar to ours in scope and approach, namely, the Ivy system [41] and the work on exploiting modularity for decidability presented in [44]. Our approach bears similarities to these other two verification techniques, but there are a number of key differences in terms of goals and methodologies.

One main focus of Ivy is on the modeling language, with a goal of making it easy to represent systems in a decidable fragment of first order logic, so as to ensure verification conditions always provide some concrete feedback in the form of counterexamples. They also discuss an interactive approach for generalization from counterexamples, that has similarities to the UPDR approach used in extensions of IC3/PDR [24]. In contrast, our work is primarily focused on different concerns e.g., we focus on compositionality as a means to provide an efficient and scalable proof methodology, and as a means to produce a more interpretable proof artifact, in addition to allowing for localized counterexample reasoning. We also view decidable modeling as an orthogonal component of the verification process that could be complementary to our approach.

The goals of the work in [44] are more similar to our own, in that they aim to use a particular type of compositional reasoning to exploit decidable subproblems when possible. This is perhaps

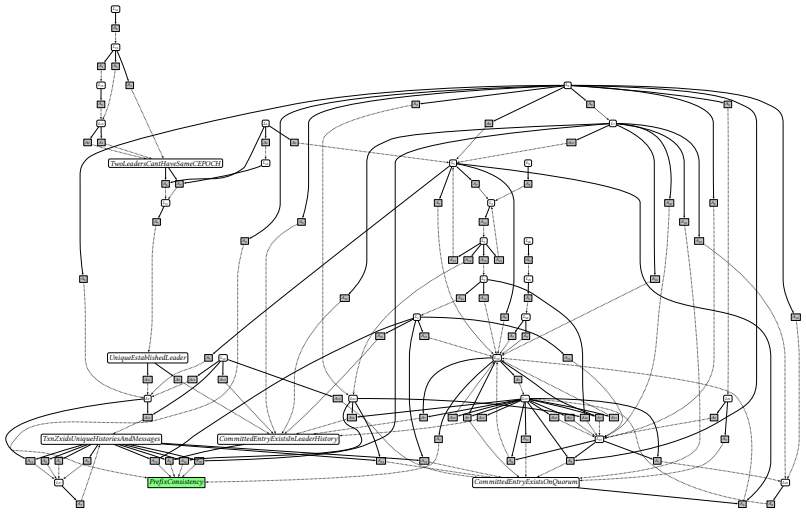


Fig. 10. *Zab* inductive proof graph.

closest to our approach in that it tries to exploit decomposition in the verification and proof process, but the decomposition notions and the way they are used are somewhat different between our work and theirs. Our goal is to define a compositional structure that is integrated into the counterexample guidance process, while also producing a single, inductive proof artifact upon completion. In future it would be interesting, however, to consider whether our notion of decomposition based on the inductive proof graph structure admits a similar kind of "decidable subproblem" property that is exploited in [44].

Concurrent Program Analysis. Our techniques presented in this paper bear similarities to prior approaches used in the proofs and analysis of concurrent programs. Our notion of inductive proof graphs is similar in nature to the *inductive data flow graph* concept presented in [10]. That work, however, is focused specifically on the verification of multi-process concurrent programs, and did not generalize the notions to a distributed setting. The procedures for verification and counterexample analysis are also different between our approach and theirs.

Our counterexample slicing technique is similar to a cone-of-influence reduction [13], as well as other *program slicing* techniques [45]. It also shares some concepts with other path-based program analysis techniques that incorporate slicing techniques [21, 22]. In our case, however, we apply it at the level of a single protocol action and target lemma, particularly for the purpose of CTI state projection.

Automated Inductive Invariant Inference. There are several recently published techniques that attempt to solve the problem of fully automated inductive invariant inference for distributed protocols, including IC3PO [11], [5], SWISS [15] and DistAI [51]. These tools, however, provide little feedback when they fail on a given problem, and the large scale protocols we presented in this paper (*AsyncRaft* and *Zab*), are of a complexity considerably higher than what modern tools in this area can solve.

Our techniques for managing large scale proof structures also bear some similarities with approaches developed for managing proof obligation queues in IC3/PDR [2, 14]. In a sense, our approach revolves around making the set of proof obligations and their dependencies explicit (and

also incorporating action-based decomposition), which can be a key factor in tuning of IC3/PDR, which often has many non-deterministic choices throughout execution.

8 CONCLUSIONS AND FUTURE WORK

We presented inductive proof decomposition, a new methodology for human-guided development of inductive invariants for large-scale protocol safety verification. In future, we are interested in exploring approaches enabled by this technique and proof structure e.g., integrating more automation by applying local syntax-guided synthesis techniques to proof graph nodes. We are also interested in understanding how these proof graph structures might be useful as a part of other automated model checking engines, and in understanding the structure of these proof graphs for additional real world protocols.

REFERENCES

[1] Barrett, C., and Tinelli, C. *Satisfiability Modulo Theories*. Springer International Publishing, Cham, 2018, pp. 305–343.

[2] Berryhill, R., Ivrii, A., Veira, N., and Veneris, A. Learning support sets in IC3 and Quip: The good, the bad, and the ugly. In *2017 Formal Methods in Computer Aided Design (FMCAD)* (2017), IEEE, pp. 140–147.

[3] Bertot, Y., and Castéran, P. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[4] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. Bounded model checking. *Adv. Comput. 58* (2003), 117–148.

[5] Bradley, A. R. SAT-Based Model Checking without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2011), VMCAI'11, Springer-Verlag, p. 70–87.

[6] Braithwaite, S., Buchman, E., Konnov, I., Milosevic, Z., Stoilkovska, I., Widder, J., and Zamfir, A. Formal Specification and Model Checking of the Tendermint Blockchain Synchronization Protocol (Short Paper). In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)* (2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[7] Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang. 1*, ICFP (aug 2017).

[8] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Woodford, D., Saito, Y., Taylor, C., Szymaniak, M., and Wang, R. Spanner: Google's Globally-Distributed Database. In *OSDI* (2012).

[9] De Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[10] Farzan, A., Kincaid, Z., and Podelski, A. Inductive Data Flow Graphs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013* (2013), R. Giacobazzi and R. Cousot, Eds., ACM, pp. 129–142.

[11] Goel, A., and Sakallah, K. 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* (Berlin, Heidelberg, 2021), Springer-Verlag, p. 131–150.

[12] Goel, A., and Sakallah, K. A. Towards an Automatic Proof of Lamport's Paxos. *2021 Formal Methods in Computer Aided Design (FMCAD)* (2021), 112–122.

[13] Gordon, M. J., Kaufmann, M., and Ray, S. The Right Tools for the Job: Correctness of Cone of Influence Reduction Proved Using ACL2 and HOL4. *J. Autom. Reason. 47*, 1 (jun 2011), 1–16.

[14] Gurfinkel, A., and Ivrii, A. Pushing to the top. *2015 Formal Methods in Computer-Aided Design (FMCAD)* (2015), 65–72.

[15] Hance, T., Heule, M., Martins, R., and Parno, B. 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)* (Apr. 2021), USENIX Association, pp. 115–131.

[16] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, Association for Computing Machinery, p. 1–17.

[17] Holzmann, G. J. The model checker SPIN. *IEEE Transactions on software engineering 23*, 5 (1997), 279–295.

[18] Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Shen, L., Tang, L., Zhou, Y., Huang, M., Wei, W., Liu, C., Zhang, J., Li, J., Wu, X., Song, L., Sun, R., Yu, S., Zhao, L., Cameron, N., Pei, L., and Tang, X. TiDB: A Raft-Based HTAP

Database. *Proc. VLDB Endow. 13*, 12 (aug 2020), 3072–3084.

[19] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (USA, 2010), USENIXATC'10, USENIX Association, p. 11.

[20] Hunt Jr, W. A., Kaufmann, M., Moore, J. S., and Slobodova, A. Industrial hardware and software verification with acl2. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 375*, 2104 (2017), 20150399.

[21] Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. Path-sensitive backward slicing. In *Static Analysis: 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings 19* (2012), Springer, pp. 231–247.

[22] Jhala, R., and Majumdar, R. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language Design and Implementation* (2005), pp. 38–47.

[23] Junqueira, F. P., Reed, B. C., and Serafini, M. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)* (2011), IEEE, pp. 245–256.

[24] Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., and Shoham, S. Property-Directed Inference of Universal Invariants or Proving Their Absence. *J. ACM 64*, 1 (mar 2017).

[25] Koenig, J. R., Padon, O., Immerman, N., and Aiken, A. First-Order Quantified Separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020), PLDI 2020, Association for Computing Machinery, p. 703–717.

[26] Koenig, J. R., Padon, O., Shoham, S., and Aiken, A. Inferring Invariants with Quantifier Alternations: Taming the Search Space Explosion. In *Tools and Algorithms for the Construction and Analysis of Systems* (Cham, 2022), D. Fisman and G. Rosu, Eds., Springer International Publishing, pp. 338–356.

[27] Konnov, I., Kukovec, J., and Tran, T.-H. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct 2019).

[28] Lamport, L. How to write a proof. *The American mathematical monthly 102*, 7 (1995), 600–608.

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

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

[31] Lamport, L. Using TLC to Check Inductive Invariance. http://lamport.azurewebsites.net/tla/inductive-invariant.pdf, 2018.

[32] Lampson, B., and Sturgis, H. Crash Recovery in a Distributed Data Storage System. *Unpublished technical report, Xerox Palo Alto Research Center* (06 1979).

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

[34] Manna, Z., and Pnueli, A. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, Heidelberg, 1995.

[35] Newcombe, C. Why Amazon Chose TLA+. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z* (2014), Springer, pp. 25–39.

[36] Nipkow, T., Wenzel, M., and Paulson, L. C. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[37] Ongaro, D. Consensus: Bridging Theory and Practice. *Doctoral thesis* (2014).

[38] Ongaro, D., and Ousterhout, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320.

[39] Ouyang, L., Huang, Y., Huang, B., Wei, H., and Ma, X. Leveraging TLA+ specifications to improve the reliability of the zookeeper coordination service. *CoRR abs/2302.02703* (2023).

[40] Padon, O., Immerman, N., Shoham, S., Karbyshev, A., and Sagiv, M. Decidability of Inferring Inductive Invariants. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2016), POPL '16, Association for Computing Machinery, p. 217–231.

[41] Padon, O., McMillan, K. L., Panda, A., Sagiv, M., and Shoham, S. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016), PLDI '16, Association for Computing Machinery, p. 614–630.

[42] Schultz, W., Dardik, I., and Tripakis, S. Formal Verification of a Distributed Dynamic Reconfiguration Protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA, 2022), CPP 2022, Association for Computing Machinery, p. 143–152.

[43] Taft, R., Sharif, I., Matei, A., VanBenschoten, N., Lewis, J., Grieger, T., Niemi, K., Woods, A., Birzin, A., Poss, R., Bardea, P., Ranade, A., Darnell, B., Gruneir, B., Jaffray, J., Zhang, L., and Mattis, P. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), SIGMOD '20, Association for Computing Machinery, p. 1493–1509.

[44] Taube, M., Losa, G., McMillan, K. L., Padon, O., Sagiv, M., Shoham, S., Wilcox, J. R., and Woos, D. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2018), pp. 662–677.

[45] Tip, F. A survey of program slicing techniques. *J. Program. Lang. 3* (1994).

[46] Vanlightly, J. raft-tlaplus: A TLA+ specification of the Raft distributed consensus algorithm. https://github.com/Vanlightly/raft-tlaplus/blob/main/specifications/standard-raft/Raft.tla, 2023. GitHub repository.

[47] Wilcox, J. R. *Compositional and Automated Verification of Distributed Systems.* PhD thesis, University of Washington, USA, 2021.

[48] Wilcox, J. R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M. D., and Anderson, T. E. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (2015), D. Grove and S. M. Blackburn, Eds., ACM, pp. 357–368.

[49] Woos, D., Wilcox, J. R., Anton, S., Tatlock, Z., Ernst, M. D., and Anderson, T. Planning for Change in a Formal Verification of the Raft Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (2016), CPP 2016, Association for Computing Machinery, p. 154–165.

[50] Yao, J., Tao, R., Gu, R., and Nieh, J. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), M. K. Aguilera and H. Weatherspoon, Eds., USENIX Association, pp. 485–501.

[51] Yao, J., Tao, R., Gu, R., Nieh, J., Jana, S., and Ryan, G. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), USENIX Association, pp. 405–421.

[52] Yu, Y., Manolios, P., and Lamport, L. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods* (Berlin, Heidelberg, 1999), L. Pierre and T. Kropf, Eds., Springer Berlin Heidelberg, pp. 54–66.