Interpretable Safety Verification of Distributed Protocols by Inductive Proof Decomposition

William Schultz
Northeastern University
Boston, MA, USA
schultz.w@northeastern.edu

Edward Ashton
Azure Research, Microsoft
Cambridge, UK
edward.ashton@microsoft.com

Heidi Howard
Azure Research, Microsoft
Cambridge, UK
heidi.howard@microsoft.com

Stavros Tripakis
Northeastern University
Boston, MA, USA
s.tripakis@northeastern.edu

Abstract

Many techniques for the automated verification of distributed protocols have been developed over the past several years, centered around automatic inference of an *inductive invariant* for proving safety. The performance of these techniques, however, can still be unpredictable and their failure modes opaque. 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 protocol verification that provides a novel, compositional approach to inductive invariant development. Our technique aims to bridge the gap between the automation provided by modern inference algorithms with the interaction and interpretability often needed in larger scale proofs. Our approach is centered around the insight that any inductive invariant can be decomposed into an *inductive proof graph*, a core data structure that we use to guide the compositional development of an inductive invariant. We present an algorithm to synthesize these graphs efficiently while also permitting interaction from a human in cases of failure to synthesize a complete proof. We present our technique and experience using it to develop an automated inductive safety proof of a large scale, asynchronous specification of the Raft consensus protocol, the first such automated proof for a distributed protocol of this complexity. We also demonstrate how these proof graphs provide insight into the structure of protocol correctness proofs, something not afforded by existing techniques.

1 Introduction

Verifying the safety of large-scale distributed 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 [4, 14, 20, 27, 34]. Moreover, critical safety and liveness bugs continue to be found in core protocols [26, 30, 33], underscoring the value of verifying these protocol designs. 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 [38, 39].

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 [10, 19, 29, 41]. Many of these tools are based on modern model checking algorithms like IC3/PDR [10, 11, 17–19], and others based on syntax-guided or enumerative invariant synthesis methods [13, 32, 40]. 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 [11, 20]. The theoretical complexity limits facing these techniques, however, limit their ability to be fully general [28] and, even in practice, the performance of these tools on complex protocols is still unpredictable, and their failure modes can be opaque.

In particular, one key drawback of these methods is that, in their current form, they are very much “all or nothing”. That is, a given problem can either be automatically solved with no manual proof effort, or the problem falls outside the method’s scope and a global failure is reported. In the latter case, little assistance is provided in terms of how to develop a manual proof or how a human can offer guidance to the tool. We believe there is significant utility in providing a smoother transition between these possible outcomes. In practice, real world, large-scale verification efforts often benefit from some amount of human interpretability and interaction i.e., a human provides guidance when an automated engine is unable to automatically prove certain properties about a design or protocol. This may involve simplifying the problem statement given to the tool, or completing some part of the tool’s proof process by hand. Recent verification efforts of industrial scale protocols often note the high amount of human effort in developing inductive invariants. Some leave human integration as future goals [3, 31], while others have 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 [11, 23].

In this paper we present *inductive proof decomposition*, a new technique for inductive invariant development that aims to address these limitations of existing approaches. Our technique utilizes the underlying compositional structure of an inductive invariant to guide its development, based on the insight that a standard inductive invariant can be decomposed into an *inductive proof graph*. This graph structure makes explicit the induction dependencies between lemmas of an inductive invariant, and their relationship to the logical transitions of a concurrent or distributed protocol. It serves as a core guidance mechanism for inductive invariant development, by making the global dependency structure apparent. In addition, the structure of the proof graph allows for localized reasoning about proof obligations, enabling focus on small sub-problems of the inductive proof rather than a large, monolithic inductive invariant.

We build a technique for automatically and efficiently synthesizing these proof graphs, thus enabling automation while preserving amenability to human interaction and interpretability. We demonstrate that these proof graphs can be presented to and interpreted directly by a human user, facilitating a concrete and effective diagnosis and interaction process, enhancing interpretability of both the final inductive proof and the intermediate results. In addition, our automated synthesis technique also takes advantage of the proof graph structure to accelerate its local synthesis tasks by computing local variable *slices* at nodes of the graph, via localized static analyses. That is, we are able to project away state variables that are irrelevant to proving a local proof obligation, allowing for both improved efficiency and interpretability.

We apply our technique to develop an inductive proof of an industrial-scale, asynchronous specification of the Raft [27] consensus protocol, demonstrating the effectiveness of our technique as well as its interpretability features. Notably, this automated verification task is for a protocol specification of considerably higher complexity than achieved by other state of the art automated protocol verification tools [11, 18, 40].

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 protocol.
- *Inductive proof decomposition*, a new compositional inductive invariant development technique that is amenable both to efficient automated synthesis and fine-grained human interaction and interpretability.
- Implementation of our technique in a verification tool, Scimitar, and an evaluation of its use in developing an inductive safety proof of a large-scale, asynchronous formal specification of the Raft [27] consensus protocol.

CONSTANTS $Node, Value, Quorum$

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

Protocol actions.

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

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

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

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

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

Safety property.

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

Figure 1. State variables, protocol actions, and safety property (*NoConflictingValues*) for the *SimpleConsensus* protocol. Initial conditions omitted for brevity.

2 Overview

To illustrate the core ideas of *inductive proof decomposition*, our inductive invariant development technique, we walk through it on a small example protocol.

Figure 1 shows a formal specification of a simple consensus protocol, defined as a symbolic transition system in TLA$^+$ style notation [21]. This protocol utilizes a simple leader election mechanism 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 1, states that no two differing values can be chosen. The protocol's specification consists of 6 state variables and 5 distinct protocol *actions*, expressed in a *guarded action* style i.e., actions are

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

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

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

$NodesVoteOnce \triangleq$
$\quad \forall n, n_i, n_j \in Node :$
$\quad \neg(n_i \neq n_j \land n \in votes[n_i] \land n \in votes[n_j])$

$Ind \triangleq$ Inductive invariant.
$\quad \land\ NoConflictingValues$ (Safety)
$\quad \land\ UniqueLeaders$
$\quad \land\ LeaderHasQuorum$
$\quad \land\ LeadersDecide$
$\quad \land\ NodesVoteOnce$
$\quad \land\ VoteRecvdImpliesVoteMsg$
$\quad \land\ VoteMsgsUnique$
$\quad \land\ VoteMsgImpliesVoted$

Figure 2. Complete inductive invariant, *Ind*, for proving the *NoConflictingValues* safety property of the *SimpleConsensus* protocol from Figure 1. Selected lemma definitions also shown.

of the form $A = Pre \land Post$, where Pre is a predicate over current state variables and $Post$ is a conjunction of update formulas where x_i' refers to the value of x_i in the next state of a transition.

Our overall goal is to verify that a given protocol like the one in Figure 1 satisfies its specified safety property. We can do this by discovering an *inductive invariant*, which is an invariant that (1) holds in all initial states of the system, is (2) closed under transitions of the protocol, and (3) implies our safety property. For example, given the protocol of Figure 1 as input, we may discover an inductive invariant such as *Ind* shown in Figure 2. *Ind* is the conjunction of the original safety property, plus 7 more *lemmas*, which strengthen this *NoConflictingValues* safety property (thus ensuring that *Ind* logically implies *NoConflictingValues*). Even for such a relatively simple protocol, an inductive invariant can be non-trivial in both size and logical complexity of its predicates.

Our technique, *inductive proof decomposition*, utilizes the underlying compositional structure of any inductive invariant to guide the development of an invariant such as the one in Figure 2. Specifically, our technique is centered around a data structure called an *inductive proof graph*, which we use to develop inductive invariants *incrementally* and *compositionally*. This data structure is amenable to automated synthesis while also facilitating fine-grained human interaction and interpretability due to its explicit compositional structure.

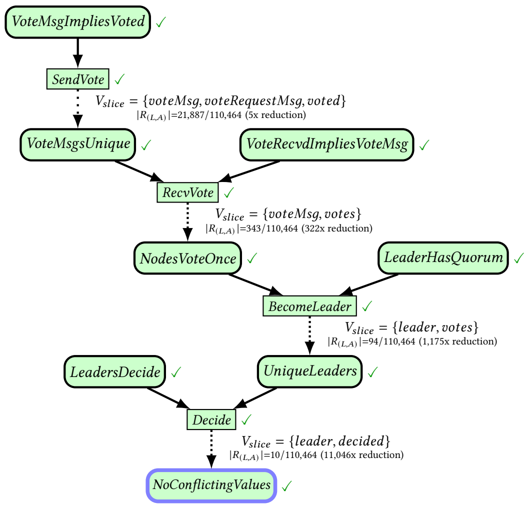

VoteMsgImpliesVoted ✓
SendVote ✓
$V_{slice} = \{voteMsg, voteRequestMsg, voted\}$
$|R_{(L,A)}|$=21,887/110,464 (5x reduction)
VoteMsgsUnique ✓
VoteRecvdImpliesVoteMsg ✓
RecvVote ✓
$V_{slice} = \{voteMsg, votes\}$
$|R_{(L,A)}|$=343/110,464 (322x reduction)
NodesVoteOnce ✓
LeaderHasQuorum ✓
BecomeLeader ✓
$V_{slice} = \{leader, votes\}$
$|R_{(L,A)}|$=94/110,464 (1,175x reduction)
LeadersDecide ✓
UniqueLeaders ✓
Decide ✓
$V_{slice} = \{leader, decided\}$
$|R_{(L,A)}|$=10/110,464 (11,046x reduction)
NoConflictingValues ✓

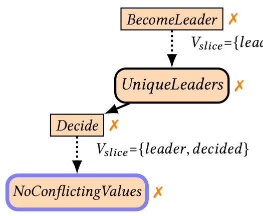
Figure 3. A complete inductive proof graph for *SimpleConsensus* protocol corresponding to the inductive invariant in Figure 2. Local variable slices are shown as V_{slice}, along with the size of the reachable state set slice at that node, indicated as $|R_{(L,A)}|$, along with the reduction factor over the full set of reachable states (of size 110,464) computed during synthesis.

A complete inductive proof graph corresponding to the inductive invariant *Ind* of Figure 2 is shown in Figure 3. The main nodes of an inductive proof graph, *lemma nodes*, correspond to lemmas of a system (so can be mapped to lemmas of a traditional inductive invariant), and the edges represent *induction dependencies* between 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. Each action node of this graph is then associated with a corresponding *inductive proof obligation*. That is, each action node A with source lemmas $L_1, ..., L_k$ and target lemma L is associated with the corresponding proof obligation

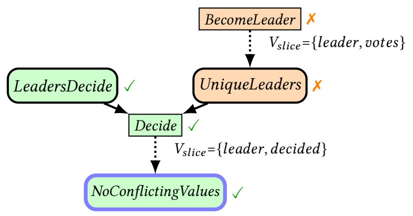
$$(L \land L_1 \land \cdots \land L_k \land A) \Rightarrow L' \qquad (1)$$

where L' denotes lemma L applied to the next-state (primed) variables. An additional, key feature of the proof graph is that each local node is associated with a *variable slice*, a subset of protocol variables sufficient to consider for discharging that node. Slices are computed from a static analysis of that node's lemma-action pair, and can be seen illustrated in Figure 3, which annotates each proof node with its variable slice.

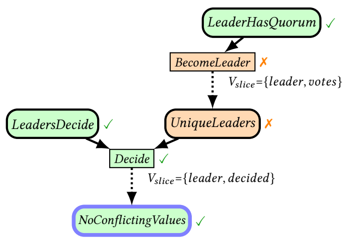
At a high level, our approach to inductive invariant development is to incrementally construct an inductive proof graph, working backwards from a specified safety property, and with specific guidance along the way. This is illustrated more concretely in Figure 4, which shows a sample of possible steps in construction of the inductive proof graph for the *NoConflictingValues* safety property of *SimpleConsensus*.



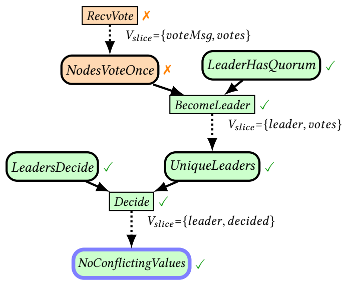
(a) In-progress inductive proof graph for *SimpleConsensus* (Step 1).



(b) Step 2.



(c) Step 3.



(d) Step 4.

Figure 4. Example progression of inductive proof graph development for *SimpleConsensus*, with variable slices shown as V_{slice}. 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.

Nodes that are unproven (shown in orange and marked with ✗), means that there are outstanding counterexamples for those inductive proof obligations. At a high level, the goal of the invariant development process is to, at each unproven node, discover support lemmas that make the lemma node inductive relative to this set of lemmas (e.g. satisfying Formula 1), discharging that proof obligation.

For example, in Figure 4a we select the unproven *Decide* node and synthesize an additional support lemma, *LeadersDecide*, to rule out counterexamples at that node that were not already eliminated by the existing support lemma, *UniqueLeaders*. After synthesizing the *LeadersDecide* lemma, this becomes a new support lemma of the *Decide* node, which is then marked as proven (shown in green and marked with ✓), as seen in Figure 4b, since all of its induction counterexamples have been eliminated by its set of support lemmas. Our algorithm continues in this fashion, next selecting the unproven *BecomeLeader* node in Figure 4b, and synthesizing the *LeaderHasQuorum* support lemma.

In Figure 4c, the current focus is on discharging the unproven *UniqueLeaders* node. Note also that the variable slice associated with this node is shown below as $\{leader, votes\}$ (2 of 6 total state variables), indicating that only those state variables must be considered when developing a support lemma, focusing the reasoning task. In addition, counterexamples to the inductive proof obligation at that node (*counterexamples to induction*) can be analyzed to guide development of a support lemma. So, a new lemma, *NodesVoteOnce*, may then be synthesized to discharge *UniqueLeaders* and added to the graph, as shown in Figure 4d. As shown there, newly synthesized support lemmas create new proof obligations to consider (e.g. via *NodesVoteOnce*), with different variable slices. The process continues until all nodes are discharged e.g., leading to a complete inductive proof graph as shown in Figure 3.

A main feature of the approach to inductive invariant development as outlined above is that it is amenable both to efficient automation and interaction and interpretability. With this in mind, we build an automated technique for synthesizing inductive proof graphs using a syntax-guided invariant synthesis technique [1, 8, 32]. The structure of the inductive proof graph and localized nature of these synthesis tasks also enables several *slicing* based optimizations, accelerating our automated synthesis routines. Crucially, due to the incremental maintenance of the proof graph during this overall synthesis procedure, we can allow fine-grained feedback and interaction from a human in the case of failure to produce a complete proof.

In the remainder of this paper, we formalize the above ideas and techniques in more detail, and present an evaluation applying our techniques to a complex, asynchronous distributed protocol.

3 Inductive Proof Graphs

Our inductive invariant development technique is based around a core logical data 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 efficient automated synthesis, and also to localized reasoning and human interpretability, as we discuss further in Sections 4 and 5.

3.1 Decomposing Inductive Invariants

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 [24]. 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 inductive proof graphs, and which decomposes an inductive invariant based on this 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

$$(S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T \Rightarrow (S \wedge L_1 \wedge \cdots \wedge L_k)' \qquad (2)$$

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

$$\begin{gathered}(S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T \Rightarrow S' \\ (S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T \Rightarrow L_1' \\ \vdots \\ (S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T \Rightarrow L_k'\end{gathered} \qquad (3)$$

If the overall invariant Ind is inductive, then each of the proof obligations in Formula 3 must be valid. That is, we say that each lemma in Ind is inductive *relative* to the conjunction of lemmas in $\{S, L_1, \ldots, L_k\}$.

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 3, $\mathcal{L}$ 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 $\mathcal{L}$.

For distributed and concurrent protocols, 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 shown in the example of Figure 1. So, each node of a lemma support graph can be 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.

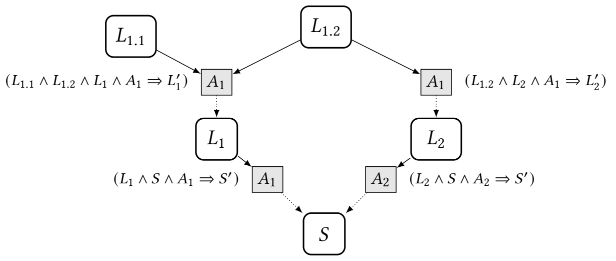

$L_{1.1}$
$L_{1.2}$
$(L_{1.1} \wedge L_{1.2} \wedge L_1 \wedge A_1 \Rightarrow L_1')$
A_1
A_1
$(L_{1.2} \wedge L_2 \wedge A_1 \Rightarrow L_2')$
L_1
L_2
$(L_1 \wedge S \wedge A_1 \Rightarrow S')$
A_1
A_2
$(L_2 \wedge S \wedge A_2 \Rightarrow S')$
S

Figure 5. Abstract inductive proof graph example, with lemma and action nodes (in gray), and associated inductive proof obligations next to each action node. Self-inductive obligations are omitted for brevity, and action to lemma node relationships are shown as incoming lemma edges.

Definition 3.1. For a system $M = (I, T)$ with $T = A_1 \vee \cdots \vee A_n$, 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.
  - $V_A = V_L \times \{A_1, \ldots, A_n\}$ 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 5 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 edges that connect action nodes to their parent lemma node, even though these edges do not appear in the formal definition.

3.2 Inductive 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 3.2 (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:

$$\left(\wedge_{\ell \in Supp_{(L,A)}} \ell\right) \wedge L \wedge A \Rightarrow L' \qquad (4)$$

Definition 3.3 (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, $\{L\} \times \{A_1, \ldots, A_n\}$, are locally valid. We alternately refer to a lemma node that is locally valid as being *discharged*.

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

Definition 3.4 (Inductive Proof Graph Validity)**.** An inductive proof graph is *valid* whenever all lemma nodes of the graph are *locally valid*.

As an example, Figure 3 shows an example of a complete inductive proof graph satisfying the validity condition, whereas Figure 4 illustrates partial proof graphs, neither of which satisfy validity.

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 3.5. *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 3.6. *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 3.5, 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. □

3.3 Local Variable Slices

A benefit of the inductive proof graph is that its structure provides a way to focus, at each graph node, on a potentially small subset of state variables that are relevant for discharging that proof node. That is, 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 make use of this general idea to compute a *variable slice* at each node, allowing us to project away any protocol 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. Figure 3 shows an example of a proof graph annotated with its variable slices at each node. 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. Like the example protocol from Figure 1, 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 3.7. 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 $\mathcal{X} = \{x_1', \ldots, x_n'\}$, we define $COI(\mathcal{X})$ simply as $COI(x_1') \cup \cdots \cup COI(x_n')$

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 3.8. 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 3.9. *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

$$Supp \wedge L \wedge A \wedge \neg L' \tag{5}$$

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 5 can be re-written as

$$Supp \wedge L \wedge Pre \wedge \neg L'[Post] \tag{6}$$

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 6 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 $Slice(L, A)$. □

Algorithm 1 Inductive proof graph synthesis.

```
1: Inputs:
2: Transition system $M = (I, T)$, safety property $S$.
3: Grammar $Preds$, reachable state set $R$.
4: procedure SynthIndProofGraph($M$, $S$, $Preds$, $R$)
5:     $(V_L, V_A, E) \leftarrow (\{S\}, \{S\} \times \{A_1, \dots, A_n\}, \emptyset)$
6:     $G \leftarrow (V_L \cup V_A, E)$                ▷ Initialize proof graph.
7:     $failed \leftarrow \emptyset$
8:     if $\forall a \in V_A : (a$ is locally valid$) \vee (a \in failed)$ then
9:         return $(G, failed)$.   ▷ Returned graph $G$ is valid if $failed = \emptyset$
10:    else
11:        Pick $(L, A) \in (V_A \setminus failed)$ where $(L, A)$ is not locally valid.
12:        $(Supp_{(L,A)}, success) \leftarrow$ SynthLocal($M$, $Preds$, $R$, $L$, $A$)
13:        if $\neg success$ then
14:            $failed \leftarrow failed \cup \{(L, A)\}$
15:            goto Line 8
16:        end if
17:        $V_L \leftarrow V_L \cup Supp$             ▷ Update the proof graph.
18:        $V_A \leftarrow V_A \cup (Supp \times \{A_1, \dots, A_k\})$
19:        $E \leftarrow E \cup (Supp \times \{(L, A)\})$
20:        goto Line 8.
21:    end if
22: end procedure
```

4 Synthesizing Inductive Proof Graphs

Our overall technique for developing inductive invariants uses the inductive proof graph as its guiding data structure. Specifically, we build an algorithm for automatically synthesizing inductive proof graphs, allowing for a smooth transition between both automation and human interaction and interpretability.

Our proof graph synthesis algorithm utilizes core synthesis ideas presented in [32], which essentially describes a technique for efficiently synthesizing monolithic inductive invariants based on a syntax-guided approach [2]. We apply similar techniques in our context to incrementally synthesize proof graphs efficiently, by running localized synthesis tasks that can also take advantage of various slicing-based optimizations. As discussed previously, incremental maintenance of the proof graph provides an effective, fine-grained interpretability and diagnosis mechanism when our automated technique does not synthesize a complete proof graph, or has made partial progress.

4.1 Our Synthesis Algorithm

At a high level, our inductive invariant inference algorithm constructs an inductive proof graph incrementally, starting from a given safety property S as its initial lemma node. It works backwards from the safety property, incrementally synthesizing support lemmas for remaining, non-discharged proof nodes.

To synthesize these support lemmas at local graph nodes, we perform a local, syntax-guided invariant synthesis routine that is based on an extension of a prior technique [32], adapted to this compositional, graph-based setting. Once all nodes of the proof graph have been discharged, the algorithm terminates, returning a complete, valid inductive proof graph. If it cannot discharge all nodes successfully, either due to a timeout or other specified resource bounds, it may return a partial, incomplete proof graph, containing some nodes that have not been discharged and are instead marked as *failed*. The overall algorithm is described formally in Algorithm 1, which we walk through and discuss in more detail below.

Formally, our algorithm takes as input a safety property S, a transition system $M = (I, T)$, and tries to prove that S is an invariant of M by synthesizing an inductive proof graph sufficient for proving S. It starts by initializing an inductive proof graph $(V_L \cup V_A, E)$ where $V_L = \{S\}$, $V_A = \{S\} \times \{A_1, \dots, A_n\}$, and $E = \emptyset$, as shown on Line 5 of Algorithm 1. From here, the graph is incrementally extended by synthesizing support lemmas and adding support edges from these lemmas to action nodes that are not yet discharged.

As shown in the main loop of Algorithm 1 at Line 11, the algorithm repeatedly selects some node of the graph that is not discharged, and runs a local inference task at that node (Line 12 of Algorithm 1). Our local inference routine for synthesizing support lemmas $Supp_{(L,A)}$, is a subroutine, SynthLocal, of the overall algorithm, and is shown separately as Algorithm 2, and described in more detail below in Section 4.2. Once the local synthesis call SynthLocal completes successfully, the generated set of support lemmas, $Supp_{(L,A)}$, is added to the current proof graph (Line 17 of Algorithm 1), and if there are remaining nodes that are not discharged, the algorithm continues. Otherwise, it terminates with a complete, valid proof graph (Line 9 of Algorithm 1).

It is also possible that, throughout execution, some local synthesis tasks fail, due to various reasons e.g., exceeding a local timeout, exhausting a grammar, or reaching some other specified execution or resource bound. In this case, we mark a node as *failed* (Line 14 of Algorithm 1), and continue as before, excluding failed nodes from future consideration for local inference. Due to our marking of nodes as locally failed, it is possible for the algorithm to terminate with some nodes that are not discharged (i.e. are marked in *failed*). We discuss this aspect further in our evaluation section where we discuss the interpretability and diagnosis capabilities of our approach.

The above outlines the execution of our algorithm at a high level. To accelerate it, however, we rely on several key optimizations that are enabled by the variable slicing computations we perform during local inference. We discuss these in more detail below and how they accelerate our overall inference procedure.

4.2 Local Lemma Synthesis with Slicing

As described above and shown in Algorithm 2, our local synthesis routine, SynthLocal, consists of a main loop that searches for candidate protocol invariants to serve as a valid set of support lemmas. Following prior syntax-guided approaches for synthesizing inductive invariants [32, 41], we utilize a set of reachable protocol states, R, to look for these invariants, and *counterexamples to induction* (CTIs) to guide selection from among these invariants those that are relevant to the current inductive proof obligation. In other words, each local synthesis task can be viewed as finding a set of predicates that act as a separator between the set of states in R and a set of local induction counterexamples, $CTIs$.

The search space for these lemma support invariants is defined by a grammar of state predicates given as input to our overall algorithm, *Preds*, and the GenLemmaInvs routine uses a set of reachable system states R to validate candidate invariants sampled from this grammar. The details of the GenLemmaInvs subroutine are largely based on techniques from existing work on monolithic inductive invariant synthesis [32]. In general, though, we search for lemma invariant candidates in increasing size order (e.g. in max number of syntactic terms) until reaching a specified search bound. Thus, the number of candidates generated by *Preds* and the size of R are the main factors impacting the performance of these local synthesis tasks, which make up the main computational work of our overall algorithm. We accelerate these tasks by making use of the local variable slices at each node to apply both *grammar slicing* and *state slicing* optimizations.

At the beginning of local synthesis at a node (L, A), we use the local variable slice, $Vars_{(L,A)}$, to prune the set of predicates in the global set *Preds*. That is, we filter out any predicates in *Preds* that do not refer to a subset of variables in $Vars_{(L,A)}$ (Line 4 of Algorithm 2). We then also compute a local projection, $R_{(L,A)} = \{\Pi_{Vars_{(L,A)}}(r) : r \in R\}$, of the reachable state set R (Line 3 of Algorithm 2), where $\Pi_{\mathcal{V}}$ is a projection function that maps a state r to a new state that only includes the variables in $\mathcal{V}$. These projections can be computed efficiently based on the full, cached set R, and can also be cached as needed during synthesis. Both the local grammar slice and state projection are then passed as inputs to our invariant enumeration routine, GenLemmaInvs. We show in our evaluation how these optimizations impact the efficiency of the synthesis procedure.

5 Evaluation

We focused our evaluation on the application of our technique to complete a large scale, highly automated inductive proof of a complex, asynchronous specification of the Raft consensus protocol [27]. We focused on this protocol benchmark since it most rigorously exercised the features and benefits of our approach e.g. related to interpretability and slicing based performance optimizations. Similarly, it served as a case study of distributed protocol verification at a scale significantly larger than other state of the art automated protocol verification tools [11, 18, 40]. To our knowledge, ours is the first automated inductive invariant synthesis effort for a specification of Raft of this complexity.

Overall, our evaluation aimed to examine the interpretability features of our approach, the effect of our slicing optimizations, as well as the experience of using our technique to develop a proof for a complex, asynchronous distributed protocol.

Implementation and Setup. We implemented our technique and synthesis algorithms described in Section 4 in a verification tool, Scimitar, which consists of approximately 6100 lines of Python code, and accepts as input protocols specified in the TLA$^+$ specification language [21]. Internally, Scimitar uses the TLC model checker [42] for most of its compute-intensive inference sub-routines, like checking candidate lemma invariants and CTI generation and elimination checking. Specifically, it uses TLC to generate counterexamples to induction for finite protocol instances using a randomized search technique [22]. Our current implementation uses TLC version 2.15 with some modifications to enable the optimizations employed in our technique, including the computation of state space caching and projections for different variable slices. We also use the TLA$^+$ proof system (TLAPS) [5] to validate the correctness of the inductive invariants inferred by our tool.

All of our experiments below were carried out on a 56-core Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz machine with 64GB of allocated RAM, and our tool was configured to use a maximum of 24 worker threads for TLC model checking and other parallelizable inference tasks.

Algorithm 2 Local support lemma synthesis.

```
1: procedure SynthLocal($M$, $Preds$, $R$, $L$, $A$)
2:     $Vars_{(L,A)} \leftarrow$ Slice($L$, $A$)                ▷ See Definition 3.8.
3:     $R_{(L,A)} \leftarrow \{\Pi_{Vars_{(L,A)}}(r) : r \in R\}$        ▷ Project $R$ to the slice.
4:     $Preds_{(L,A)} \leftarrow \{p \in Preds : Vars(p) \subseteq Vars_{(L,A)}\}$    ▷ Grammar slice.
5:     $Supp_{(L,A)} \leftarrow \emptyset$
6:     $CTIs \leftarrow$ CTIs($M, L, A$)      ▷ Find states s.t. $\neg(Supp_{(L,A)} \land L \land A \land L')$.
7:     while $CTIs \neq \emptyset$ do
8:         $Invs \leftarrow$ GenLemmaInvs($M$, $Vars_{(L,A)}$, $Preds_{(L,A)}$, $R_{(L,A)}$)
9:         if $\exists A \in Invs : A$ eliminates some CTI in $CTIs$ then
10:            Pick $L_{max} \in Invs$ that eliminates the most CTIs from $CTIs$.
11:            $Supp_{(L,A)} \leftarrow Supp_{(L,A)} \cup \{L_{max}\}$
12:            $CTIs \leftarrow CTIs \setminus \{s \in CTIs : s \not\models L_{max}\}$
13:        else
14:            either goto Line 8
15:            or return ($Supp_{(L,A)}$, False)       ▷ Couldn't eliminate CTIs.
16:        end if
17:    end while
18:    return ($Supp_{(L,A)}$, True)          ▷ Success: eliminated all CTIs
19: end procedure
```

State Variables	**Actions**
currentTerm	*RequestVote*
state	*UpdateTerm*
votedFor	*HandleRequestVoteRequest*
log	*HandleRequestVoteResponse*
commitIndex	*BecomeLeader*
votesGranted	*ClientRequest*
nextIndex	*AppendEntries*
matchIndex	*AcceptAppendEntries*
requestVoteRequestMsgs	
requestVoteResponseMsgs	
appendEntriesRequestMsgs	
appendEntriesResponseMsgs	

Figure 6. Actions and variables of the *AsyncRaft* protocol specification. The message state variables (suffixed with *Msgs*) are sets that model network messages sent between servers of the protocol.

Raft Protocol Specification. For our evaluation we use a TLA$^+$ specification of an asynchronous, message-passing version of the Raft consensus protocol [27], which we refer to as *AsyncRaft*. Our specification is based on the original specification published in the Raft dissertation [25] and a similar, updated variant [37]. Some additional modifications were made to make the specification amenable to model checking and synthesis tasks. For example, we also altered the original specification slightly to be written in a paradigm we refer to as a *universal* message passing style. Essentially, this specification approach allows us to keep the format of data structures between node state and message state nearly uniform, which helps aid in the understandability of the protocol, and also makes certain structural features of lemma invariants more apparent as we discuss below in Section 5.3.

Our specification models all core aspects of elections and log replication of the Raft protocol, and contains 8 core actions, 12 state variables, and consists of approximately 600 lines of TLA$^+$ code. The main actions and variables of the specification are summarized in Figure 6. The *RequestVote*, *UpdateTerm*, *HandleRequestVoteRequest*, *HandleRequestVoteResponse*, and *BecomeLeader* actions are related to the leader election process, while the *ClientRequest*, *AppendEntries* and *AcceptAppendEntries* actions are related to log replication. The specification is also parameterized by a set *Server*, which defines the finite set of nodes operating in the protocol.

We note that this *AsyncRaft* specification benchmark is of a complexity considerably greater than those tested by recent automated invariant inference techniques. For example, one of the most recent tools, DuoAI [40], reports the LoC of the largest protocol tested as 123 lines of code in the Ivy language [29], which is of a similar abstraction level to TLA$^+$. Our specification of Raft is around 600 lines of TLA$^+$, so we view our evaluation as examining scalability of our technique on protocols that are notably more complex than those tested by existing approaches.

$$
\begin{aligned}
&\mathit{ElectionSafety} \triangleq \\
&\quad \forall s, t \in \mathit{Server} : \\
&\quad (\mathit{state}[s] = \mathit{Leader} \land \mathit{state}[t] = \mathit{Leader} \land s \neq t) \Rightarrow \\
&\qquad (\mathit{currentTerm}[s] \neq \mathit{currentTerm}[t])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{LogMatching} \triangleq \\
&\quad \forall s, t \in \mathit{Server} : \forall i \in \text{DOMAIN } \mathit{log}[s] : \\
&\quad (\exists j \in \text{DOMAIN } \mathit{log}[t] : i = j \land \mathit{log}[s][i] = \mathit{log}[t][j]) \Rightarrow \\
&\qquad (\mathit{SubSeq}(\mathit{log}[s], 1, i) = \mathit{SubSeq}(\mathit{log}[t], 1, i))
\end{aligned}
$$

Figure 7. Top level safety property, *LogMatching*, and key lemma from the *AsyncRaft* proof graph of Figure 8.

5.1 Results

Using our verification tool, Scimitar, we synthesized an inductive proof graph for a core, high level safety property of our Raft specification, the *LogMatching* property, which is a core property in the overall correctness proof of Raft, and whose definition is shown in Figure 7. We were able to synthesize a proof graph for this property in approximately 21 hours, consisting of 33 total lemma nodes, as shown in Figure 8.

The base grammar used for this synthesis task consists of 173 predicates, which are filtered down at various points based on the current variable slice. For our synthesis run, we sampled a reachable state set R by simulating protocol traces using the TLC model checker, generating 100,000 traces, each with a max depth of 100. These include traces sampled from finite models of the protocol for both 3 and 5 node configurations for the *Server* parameter, and generated in total 561,540 unique reachable states in R. These states are stored upfront and state slice projections then computed and cached as needed. For protocols of this size, it is not feasible to exhaustively sample all reachable states, so we used this sampling approach as is done in other syntax-guided synthesis approaches [32, 41], and we found this to be effective in discovering correct candidate lemma invariants.

In the following sections, we examine in more depth the interpretability features of this synthesized proof graph, as well as the impact of slicing optimizations and proof development experience.

5.2 Impact of Slicing Optimizations

Scaling our technique to a protocol of this size was a core challenge, and our slicing optimizations were critical to enabling this. We found that trying to carry out invariant synthesis on the base protocol quickly became infeasible, due to scalability concerns of checking many candidate invariants over the entire, non-projected state space. To more concretely

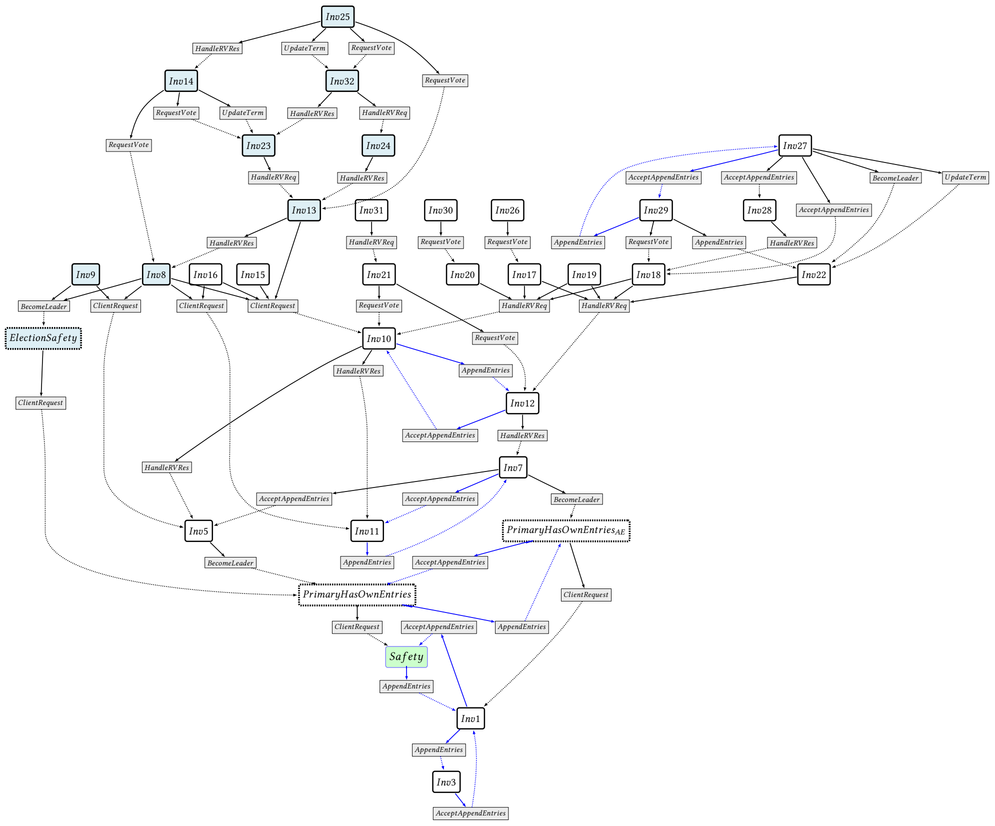


Figure 8. Our synthesized inductive proof graph for the *LogMatching* safety property (labeled as *Safety* in green) of our *AsyncRaft* specification. Key lemmas named and annotated with dotted outlines. Support subgraph for *ElectionSafety* filled in light blue, and induction cycle edges highlighted in blue. Some protocol action names abbreviated for brevity.

quantify the impact of these slicing optimizations, we examined the distribution of slice sizes relative to the raw size of the reachable sample set $|R|$. We measured the size of state slices computed across the entire synthesis run for the *LogMatching* property, and these metrics are summarized in Figure 9.

We note that, in our tool implementation, when computing a cached set of states for a given variable slice V_{slice}, we may also compute even finer-grained state slice caches, based on the set of variables that appear in candidate invariants we sampled. For example, if a slice contains 5 variables, but a subset of candidate invariants refer to only 2 / 5 distinct variables from this slice, we will also compute a slice cache for this 2 variable projection, and use that when checking that candidate invariant subset. These details are reflected in Figure 9, where the median state slice provides a roughly 16x reduction in size over the full sampled reachable state set. Moreover, there are some state slice sets with much more significant reductions e.g. the variable slice $\{currentTerm, state, votesGranted\}$ contains 3483 unique states, a 161x reduction of the full set size $|R| = 561,540$ used in this synthesis run.

Attempting to run these synthesis tasks without slicing optimizations quickly encountered timeouts over 24-hour periods, making it infeasible to complete any meaningful proofs for this protocol without these optimizations enabled.

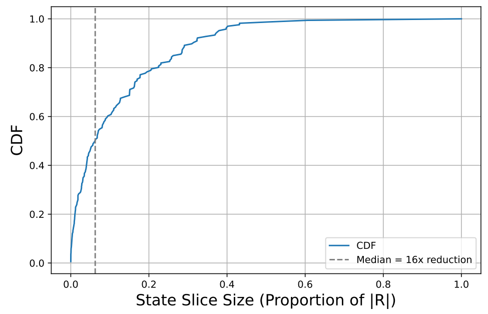


Figure 9. Cumulative distribution of state slice sizes as proportion of sampled reachable state set R, for synthesis run of *LogMatching* proof graph shown in Figure 8. Total of 166 computed slices, with median state slice size shown as roughly 16x reduction over the full size of R.

5.3 Interpretability

While our proof graph structure enables various optimizations and efficiency improvements, from an interpetability perspective, it also offers deeper insights into the underlying inductive correctness proof, and how it relates to the protocol and its behavior.

Hierarchical Structure. As illustrated in Figure 8, one key, large-scale feature of the proof graph that is not made apparent in other, monolithic inductive invariant representations is a natural, emergent hierarchical structure. This structure also reflects an intuitive understanding of the Raft protocol's behavior and correctness proof.

Most notably, this is reflected in the *ElectionSafety* support subgraph, whose lemmas are highlighted in light blue in Figure 8, and supports the *PrimaryOwnsEntries* lemma node via the *ClientRequest* action. The *ElectionSafety* lemma is a key property of Raft (shown in Figure 7), stating that two leaders cannot be elected in the same term. Generally, maintenance of this property is relied upon by the other, log replication aspects of the protocol, but is a relatively disjoint logical sub-component. This aspect is reflected in the graph structure, where most lemmas in this support subgraph do not contain interactions with other lemmas in the rest of the proof.

Message Induction Cycles. Another novel interpretability feature of this graph structure is what we refer to as *message induction cycles*. This occurs as a repeated pattern in over 6 instances in this graph (cycle edges highlighted in blue), and it occurs where certain invariants must hold both on a local server and also when its state is sent into the network via a message.

For example, the highlighted *PrimaryOwnsEntries* lemma node, which is a key supporting lemma of the top level safety property, states that if a log entry exists in a given term T, then a primary in term T must have this entry in its own log. As seen in the proof graph, an induction cycle exists between this lemma and the node labeled as $PrimaryHasOwnEntries_{AE}$. This pattern also repeats at nearby nodes with cycles of *Inv7* to *Inv11,* and *Inv10* to *Inv12.*

These local cycles form due to the message-passing nature of the protocol, and we note that these patterns bear similarities to recent observations that inductive invariant lemmas for distributed protocols often fall into a standard taxonomy [43], and some invariants may be automatically derived from others. For example, this is illustrated in Figure 10, where the full definition of the *PrimaryOwnsEntries* lemma node can be seen as syntactically related to the $PrimaryHasOwnEntries_{AE}$ lemma node, i.e. the invariants can be stated nearly identically, expect substitution that occurs for the *log* variable with the *m.mlog* variable which stores the state of a local log that sent a message into the network. These cycles appear at several locations in the graph, and help to clarify the underlying structure of the proof e.g. those lemma invariants which are fundamental to correctness and others which may be derived from neighboring lemmas.

$$
\begin{aligned}
&\mathit{PrimaryOwnsEntries}_{AE} \triangleq \\
&\quad \forall s \in \mathit{Server} : \forall m \in \mathit{appendEntriesRequestMsgs} : \\
&\quad (\mathit{state}[s] = \mathit{Leader}) \Rightarrow \\
&\quad (\nexists k \in \text{DOMAIN } m.\mathit{mlog} : \\
&\quad\quad \wedge\ m.\mathit{mlog}[k] = \mathit{currentTerm}[s] \\
&\quad\quad \wedge\ \nexists \mathit{ind} \in \text{DOMAIN } \mathit{log}[s] : (\mathit{ind} = k \wedge \mathit{log}[s][k] = m.\mathit{mlog}[k]))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{PrimaryOwnsEntries} \triangleq \\
&\quad \forall s, t \in \mathit{Server} : \\
&\quad (\mathit{state}[s] = \mathit{Leader}) \Rightarrow \\
&\quad (\nexists k \in \text{DOMAIN } \mathit{log}[t] : \\
&\quad\quad \wedge\ \mathit{log}[t][k] = \mathit{currentTerm}[s] \\
&\quad\quad \wedge\ \nexists \mathit{ind} \in \text{DOMAIN } \mathit{log}[s] : (\mathit{ind} = k \wedge \mathit{log}[s][k] = \mathit{log}[t][k]))
\end{aligned}
$$

Figure 10. Lemma pair from one *message induction cycle* in the *AsyncRaft* proof graph. It is clear to observe the underlying relationship between these two lemmas and their syntactic relationship. The former can be viewed nearly the same as the latter, accounting for substitution of the *log* variable with the *m.mlog* variable which stores the state of a local log that sent a message into the network.

5.4 Proof Development Experience

An overall strategy of our technique was to let automation complete as much proof work as possible, and let a user step in only when necessary to guide the synthesis process in a localized manner. This is a key feature of our overall technique, and one we found useful in our proof development experience i.e. we see this interaction element as a key value of our approach. That is, the implementation of our technique is highly automated, though it can allow for fine-grained diagnosis of synthesis failures when certain sub-portions of the graph are slow or difficult to synthesize.

We encountered this at a few points during initial development, and the interpretability aspects of our technique aided us in assisting the tool with extra information to facilitate convergence. For example, we encountered some cases where timeouts occurred on particular action nodes of the graph, while large subsections of the remaining graph were successfully synthesized.

In one example, in the case of *Inv12* in our proof graph, we found that, at initial proof graph synthesis attempts, difficulty to synthesize all support lemma to fully discharge its *HandleRVReq* support action. Other sections of the proof graph, however, were synthesized successfully. We were able to manually inspect this proof obligation and the outstanding CTIs that could not be automatically discharged. After some manual examination, we determined that a helper predicate that allowed lemmas to reason about prefix comparison between logs was needed to fully discharge this proof obligation. Addition of this predicate and testing it on this local proof obligation proved successful in discharging the remaining proof obligations.

In general, we found the proof graph structure a significant aid to developing this inductive proof with both automation and guidance. Without the automation and interpretability features of our technique, we do not believe it would have been possible to efficiently get a proof of this scale to go through at this level of automation.

6 Related Work

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 [10], SWISS [13] DuoAI [40, 41], and others [32]. These tools, however, provide little feedback when they fail on a given problem, and the large scale protocols we presented in this paper are of a complexity considerably higher than what existing modern tools in this area can solve.

More broadly, there exist many prior techniques for the automatic generation of program and protocol invariants that rely on data driven or grammar based approaches. Houdini [9] and Daikon [6] both use enumerative checking approaches to discover program invariants, while FreqHorn [8] tries to discover quantified program invariants about arrays using an enumerative approach that discovers invariants in stages and also makes use of the program syntax.

Interactive and Compositional Verification. There is other prior work that attempts to employ compositional and interactive techniques for safety verification of distributed protocols, but these typically did not focus on presenting a fully automated and interpretable inference technique. For example, the Ivy system [29] and additional related work on exploiting modularity for decidability [35].

In the Ivy system [29] one main focus 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 [17]. 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 automated inference technique, and as a means to produce a more interpretable proof artifact, in addition to allowing for localized counterexample reasoning and slicing. They also do not present a fully automated inference technique, as we do. Additionally, we view decidable modeling as an orthogonal component of the verification process that could be complementary to our approach.

Concurrent Program Analysis. Our techniques presented in this paper bear similarities to prior approaches used in the analysis and proofs of concurrent programs. Our notion of inductive proof graphs is similar to the *inductive data flow graph* concept presented in [7]. That work, however, is focused specifically on the verification of multi-process concurrent programs, and did not generalize the notions to a distributed setting. Our procedures for inductive invariant inference and our slicing optimizations are also novel to our approach.

Our slicing techniques are similar to cone-of-influence reductions [12], as well as other *program slicing* techniques [36]. It also shares some concepts with other path-based program analysis techniques that incorporate slicing techniques [15, 16]. In our case, however, we apply it at the level of a single protocol action and target lemma, particularly for the purpose of accelerating syntax-guided invariant synthesis tasks.

7 Conclusions and Future Work

We presented *inductive proof decomposition*, a new technique for inductive invariant development of large scale distributed protocols. Our technique both improves on the scalability of existing approaches by building an inference routine around the inductive proof graph, and this structure also makes the approach amenable to interpretability and failure diagnosis. In future, we are interested in exploring new approaches and further optimizations enabled by our technique and compositional proof structure. We would also like to explore and understand the empirical structure of these proof graphs on a wider range of larger and more complex real world protocols, and to understand the structure of inductive proof graphs with respect to protocol refinement.

Furthmore, we view these proof graph structures as a potentially promising way to bridge the gap between formal, verified induction proofs and the pen and paper style proofs often written in standard distributed systems protocol literature. These structures make apparent much of the underlying structure of formal proofs, that perhaps in many cases maps more intuitively to the structure that would be developed when writing a careful, pen and paper protocol safety proof.

References

[1] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design* (2013), pp. 1–8.

[2] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design* (2013), pp. 1–8.

[3] 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.

[4] 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).

[5] Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., and Vanzetto, H. TLA+ Proofs. *Proceedings of the 18th International Symposium on Formal Methods (FM 2012), Dimitra Giannakopoulou and Dominique Mery, editors. Springer-Verlag Lecture Notes in Computer Science 7436* (January 2012), 147–154.

[6] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon system for dynamic detection of likely invariants. *Science of computer programming 69*, 1-3 (2007), 35–45.

[7] 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.

[8] Fedyukovich, G., and Bodík, R. Accelerating Syntax-Guided Invariant Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems* (Cham, 2018), D. Beyer and M. Huisman, Eds., Springer International Publishing, pp. 251–269.

[9] Flanagan, C., and Leino, K. R. M. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity* (Berlin, Heidelberg, 2001), FME '01, Springer-Verlag, p. 500–517.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Howard, H., Malkhi, D., and Spiegelman, A. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)* (Dagstuhl, Germany, 2017), P. Fatourou, E. Jiménez, and F. Pedone, Eds., vol. 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 25:1–25:14.

[15] 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.

[16] 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.

[17] 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).

[18] 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.

[19] 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.

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

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

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

[23] 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).

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

[25] Ongaro, D. raft.tla: A tla+ specification of the raft consensus algorithm. https://github.com/ongardie/raft.tla, 2014. Accessed: 2025-05-07.

[26] Ongaro, D. Bug in single-server membership changes. https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J, jul 2015.

[27] 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.

[28] 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.

[29] 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.

[30] Pîrlea, G. Protocol bugs list. https://github.com/dranov/protocol-bugs-list, 2020. Accessed: 2024-09-17.

[31] 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.

[32] Schultz, W., Dardik, I., and Tripakis, S. Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. In *2022 Formal Methods in Computer-Aided Design (FMCAD)* (2022), IEEE, pp. 273–283.

[33] Sutra, P. On the correctness of egalitarian paxos. *CoRR abs/1906.10917* (2019).

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] Zhang, T. N., Hance, T., Kapritsos, M., Chajed, T., and Parno, B. Inductive Invariants That Spark Joy: Using Invariant Taxonomies to Streamline Distributed Protocol Proofs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 837–853.