Interpretable Safety Verification of Distributed Protocols by Inductive Proof Decomposition

Anonymous

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 approach to protocol verification that provides a compositional, interactive 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 large 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 applying it to develop inductive safety proofs of several complex protocols, including a large scale, asynchronous specification of the Raft consensus protocol, beyond the capabilities of modern automated verification techniques. We also demonstrate how these proof graphs provide insight into the structure of a protocol's correctness proof, something not afforded by existing approaches.

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 [6, 18, 41]. Moreover, critical safety and liveness bugs continue to be found in core protocols [32, 36, 40], 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 [45, 46].

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 [13, 23, 35, 48]. Many of these tools are based on modern model checking algorithms like IC3/PDR [13, 14, 21–23], and others based on syntax-guided or enumerative invariant synthesis methods [17, 38, 47]. 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 [14, 25]. The theoretical complexity limits facing these techniques, however, limit their ability to be fully general [34] 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 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, 37], 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 [14, 28].

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 a user to 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 inductive invariants of several large-scale distributed and concurrent protocol specifications, including an industrial-scale specification of the Raft [33] consensus protocol, demonstrating the effectiveness of our technique. We also provide an empirical evaluation of the interpretability and interaction features of our method.

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 empirical evaluation on several distributed protocols, including a large-scale specification of the Raft [33] consensus protocol.

CONSTANTS $Node, Value, Quorum$

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

Protocol actions.

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

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

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

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

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

Safety property.

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

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. 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

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

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

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

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

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

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

i.e., actions 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 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, the inductive invariant is 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.

A complete inductive proof graph corresponding to the

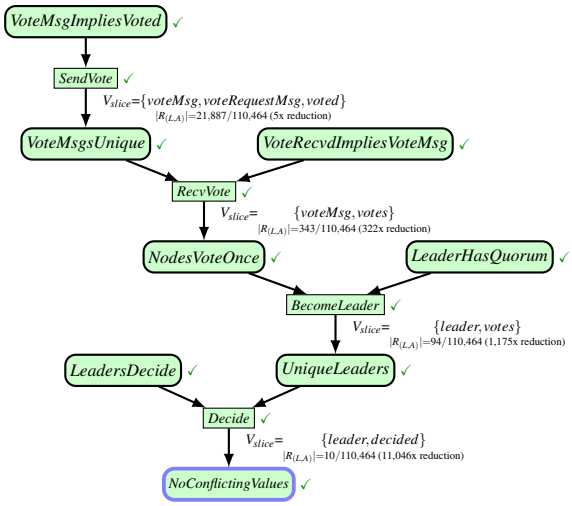


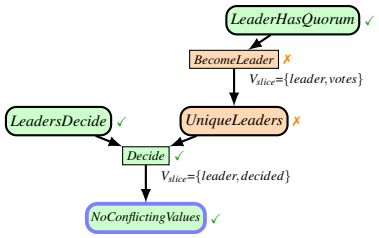
Figure 3: A complete inductive proof graph for *SimpleConsensus* protocol corresponding to the inductive invariant in Figure 2. Local variable slices are shown as V_{slice}, along with the size of the reachable state set slice at that node, indicated as $|R_{(L,A)}|$, along with the reduction factor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

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 $L_1, \ldots, L_k$ and target lemma L is associated with the corresponding proof obligation

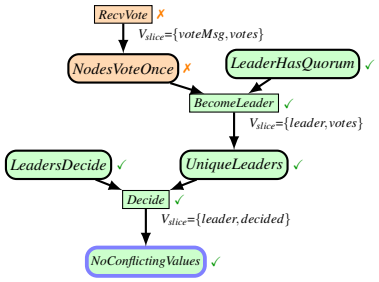
$$(L \wedge L_1 \wedge \cdots \wedge L_k \wedge A) \Rightarrow L' \tag{1}$$

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*. Nodes that are unproven (shown in orange and marked with ✗), means



(a) In-progress proof graph (Step 1).



(b) In-progress proof graph (Step 2).

Figure 4: Example progression of inductive proof graph development for *SimpleConsensus*. 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.

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, 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 examined 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 4b. 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 from a human user. With this in mind, we build an automated technique for synthesizing inductive proof graphs using a syntax-guided invariant synthesis technique [1, 10, 38]. The structure of the inductive proof graph and localized nature of these synthesis tasks also enables several *slicing* based optimizations, accelerating our automated synthesis routine. 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 several complex distributed and concurrent protocols.

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 [29]. 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

$$(S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T \Rightarrow (S \wedge L_1 \wedge \cdots \wedge L_k)' \tag{2}$$

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

$$\begin{aligned} (S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T &\Rightarrow S' \\ (S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T &\Rightarrow L_1' \\ &\vdots \\ (S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T &\Rightarrow L_k' \end{aligned} \tag{3}$$

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.

Definition 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.

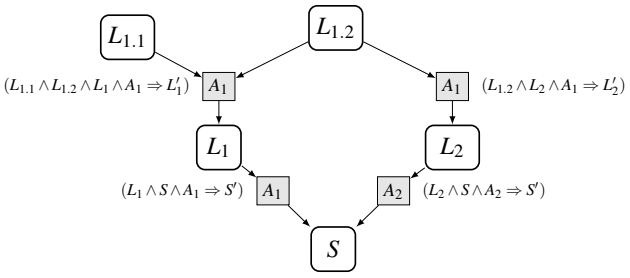


Figure 5: Abstract inductive proof graph example, with lemma and action nodes (in gray), and associated inductive proof obligations next to each action node. Self-inductive obligations are omitted for brevity, and action to lemma node relationships are shown as incoming lemma edges.

Definition 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:*

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

Definition 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 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 satsifying 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 1. *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 2. *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 1, 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. □

3.2.1 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, 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 2 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.

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 5. *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 6. *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. *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: $(V_L, V_A, E) \leftarrow (\{S\}, \{S\} \times \{A_1, \ldots, A_n\}, \emptyset)$
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: $V_A \leftarrow V_A \cup (Supp \times \{A_1, \ldots, A_k\})$
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. We build an algorithm for automatically synthesizing inductive proof graphs, allowing for a smooth transition between both (1) automation and (2) human interaction and interpretability.

Our proof graph synthesis algorithm extends ideas from previously explored inductive invariant synthesis techniques [10, 38], applying them in our context to incrementally synthesize proof graphs efficiently, by running localized synthesis tasks that 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 by synthesizing support lemmas for remaining, un-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 [38], 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 $V_L = \{S\}$, $V_A = \{S\} \times \{A_1, \ldots, A_n\}$, 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. Prior syntax-guided approaches [38, 48] for synthesizing inductive invariants utilize a set of reachable protocol states, R, to look for these invariants, and *counterexamples to induction* (CTIs) to guide selection from

Algorithm 2 Local support lemma synthesis.

1: **procedure** SYNTHLOCAL(M, *Preds*, R, L, A)
2: $Vars_{(L,A)} \leftarrow$ SLICE(L, A) ▷ See Definition 6.
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. $\neg(Supp_{(L,A)} \wedge L \wedge A \wedge L')$.
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**

among these invariants those that are relevant to the current inductive proof obligation. We adopt a similar approach in the context of each local proof node synthesis task.

The search space for these candidate 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. In general, we search for 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 simply 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, $R_{(L,A)}$, of the reachable state set R (Line 3 of Algorithm 2), projecting out any variables absent from the local variable slice $Vars_{(L,A)}$ (Line 3 of Algorithm 2). We assume these projections can be computed efficiently, and could in theory be done upfront when R is first generated. 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 can have a significant impact on the efficiency of the synthesis procedure, since we can often prune out large portions of the state and grammar space.

5 Evaluation

We evaluated our technique to understand (1) how our automated synthesis technique performs, (2) to examine the interpretability of the generated proof graphs, and also (3) how our approach allowed us to develop an inductive proof for a large, complex distributed protocol when some amount of interaction was needed.

To evaluate (1) and (2), we test our technique on a set of distributed and concurrent protocols in a range of complexities, up to distributed protocol specifications considerably larger than those previously solved by other existing tools. For testing (3) we test a large, asynchronous, message-passing specification of the Raft consensus algorithm [33], which allowed us to evaluate (2) as well. To our knowledge, ours is the first automated inductive invariant synthesis effort for a specification of Raft of this complexity.

Implementation and Setup Our technique is implemented in our verification tool, SCIMITAR, which consists of approximately 6100 lines of Python code, and accepts as input protocols specified in the TLA+ specification language [26]. Internally, SCIMITAR uses the TLC model checker [49] 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 [27]. Our current implementation uses TLC version 2.15 with some modifications to enable the optimizations employed in our technique. We also use the TLA+ proof system (TLAPS) [7] 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. We configured our tool to use a maximum of 24 worker threads for TLC model checking and other parallelizable inference tasks.

Benchmarks We used our tool to develop inductive invariants for establishing core safety properties of 6 protocol benchmarks. These protocols are summarized in Table 1, along with various statistics about the specifications and invariants. 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. Our aim in this benchmark was to evaluate our tool on a range of protocol complexities, to understand how it performs against existing techniques for smaller protocols, and then to examine both its performance gains on larger protocols, to examine the scale of protocols it could solve that existing tools could not.

The *TwoPhase* benchmark is a high level specification of the two-phase commit protocol [16], and *SimpleConsensus* is the consensus protocol presented in Section 2. The *Bakery* benchmark is a specification of Lamport's Bakery algorithm

for mutual exclusion [24], which has only recently had attempts at automated verification [12]. We also test a specification of a concurrent cache coherence protocol, *GermanCache*, which has been used as a complex and challenging verification benchmark in past work [4].

The largest of our benchmarks is an industrial scale specification of the Raft consensus protocol [33]. The specification we use is based on a model similar to the original Raft formal specification [31, 44], and models asynchronous message passing between all nodes and fine-grained local state. We verify two core safety properties of the protocol. Namely, *ElectionSafety*, stating that no two leaders can be in the same term, and another higher level lemma, *PrimaryOwnsEntries*, stating that leader servers in Raft should always contain log entries they created in their term.

We note that our largest benchmark specification, for *AsyncRaft*, is 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 our automation and interpretability features. For example, even in a recent approach, DuoAI [47] reports the LoC of the largest protocol tested as 123 lines of code in the Ivy language [35], which is of a similar abstraction level to TLA+. Our largest specification of Raft is over 500 lines of TLA+. Thus, we view our benchmarks as examining scalability of our technique on protocols that are notably more complex than those tested by existing tools.

Note that all protocols tested are parameterized, meaning that they are typically infinite-state, but have some fixed set of parameters that can be instantiated with finite parameters e.g. the set of processes or servers. Our synthesis algorithm runs using finite instantiations of protocol parameters, but our grammar templates are general enough to infer invariants that are valid for all instantiations of the protocol. Once synthesized by our tool, we validate the correctness of the inductive invariants using the TLC model checker and the TLA+ proof system [7].

Results Summary 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, number of lemmas in each proof graph, etc. We compare against another state of the art inductive invariant inference tool, *endive*, which was presented in [38], since it both accepts specifications in TLA+ and is also based on similar syntax-guided synthesis technique with similar input parameters. Thus, it makes a good candidate for comparisons since it has both performed strongly on modern distributed protocol inference benchmarks and is also the most analogous to our tool in terms of input and approach. To our knowledge, we are not aware of any other existing tool that can solve the largest benchmarks we test.

At a high level, the results can be understood as falling into roughly three distinct qualitative classes of performance. At the smallest protocol level, for benchmarks *TwoPhase* and *SimpleConsensus*, our approach successfully finds an inductive invariant, but its runtime is of comparable, albeit slower, performance than the *endive* tool, the baseline approach from [38]. We view this an expected artifact of our technique and implementation, which is optimized for scalability, at the cost of some upfront overhead when caching state slices, etc. The main goal for these smaller protocols, though, was to simply verify that our tool performs within a similar class of performance as existing approaches, and also to examine the generated proof graphs.

In the next larger class of protocols, including *Bakery* and *GermanCache*, our tool is able to solve both of these benchmarks whereas *endive* fails to solve any within a 16 hour timeout. These protocols are significantly larger than the previous benchmarks, as can be seen, for example, by their reachable state sets $|R|$, and the number of lemmas in each synthesized proof i.e., both with ≥ 20 lemmas, with *GermanCache* having 46 lemmas in its inductive proof graph.

We observed that our approach is able to leverage the slicing optimizations effectively to achieve these performance improvements on these difficult benchmarks. Note that in our implementation we in some cases compute slices at an even finer grained level than the full variable slice computed at a node, allowing for even greater acceleration. For example, if we check a set of properties that only refers to subset of variables in a local slice, we can use an even smaller state slice projection for those checks. For the *GermanCache* model, though $|R|$ is 1,663,875, the median state set size is under 1% of this full state set. We observe these slicing reductions similarly for *Bakery*, whose median state set sizes is under 2% of the reachable state set. Both of their median grammar slices are also near half of the full grammar size, which has a significant impact when searching over all candidate invariants generated by the predicates in the grammar.

The *AsyncRaft* protocol is the most complex benchmark we test, and we were able to synthesize an inductive proof graph for two high level safety properties, *ElectionSafety* (AsyncRaft$_{ES}$) and *PrimaryOwnsEntries* (AsyncRaft$_{PO}$), with both having median state sets under a few percent of the total reachable set, and similarly reduced grammar slices. We discuss the structure of those developed proof graphs and qualitative aspects of our experience developing them in more detail below in Section 5.1.

Examining Interpretability As a concrete examination of the interpretability of our method, Figure 6 shows a complete synthesized proof graph for the *TwoPhase* benchmark. This benchmark is a specification of the classic two-phase commit protocol [16], where a transaction manager aims to achieve agreement from a set of resource managers on whether to *commit* or *abort* a transaction. The proof graph shows action labels and omits full definitions of each lemma node, to illustrate the overall structure more clearly. This proof graph

						Scimitar				endive
Benchmark	**LoC**	$\|R\|$	*Vars*	|A|	$\|Preds\|$	# Lemmas	Time	$\|R\|^{\sim}_{(L,A)}$	$\|Preds\|^{\sim}_{(L,A)}$	Time
SimpleConsensus	108	110,464	6	5	25	9	748	0.1%	56%	416
TwoPhase	195	288	6	7	18	12	348	5.9 %	66 %	173
GermanCache [4]	210	1,663,875	11	12	38	46	17165	0.3%	47 %	timeout
Bakery [24]	234	6,016,610	6	7	88	20	15854	1.2 %	50 %	timeout
AsyncRaft$_{ES}$	583	2,594,148	12	9	127	9	7088	7.1 %	34 %	timeout
AsyncRaft$_{PO}$	583	2,598,265	12	9	127	37	33735	0.7%	31 %	timeout

Table 1: Protocols used in evaluation and metrics on their specifications and inductive proof graphs. The |*R*| column reports the size of the set of explored reachable states used during inference, $|Preds|$ is the number of predicates in the base grammar, and *Vars* and *A*, respectively, show the number of state variables and actions in the specification. *Time* shows the time in seconds to synthesize an inductive invariant. The (endive) column represents the baseline approach based on the technique of [38], when run with the same relevant parameters. A *timeout* entry indicates no invariant found after a 16 hour timeout. $|R|^{\sim}_{(L,A)}$ shows the median of state slice sizes computed during synthesis as a percentage of |*R*|, similarly for $|Preds|^{\sim}_{(L,A)}$.

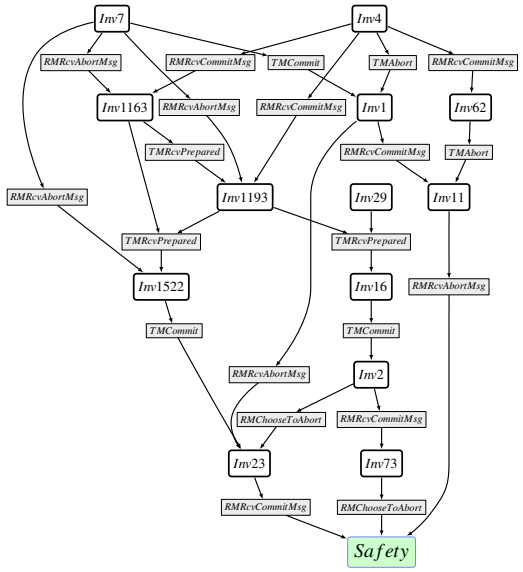


Figure 6: *TwoPhase* inductive proof graph for establishing safety property that decisions must be consistent across nodes. Excerpt of lemmas shown in Figure 7.

$$Safety \triangleq \forall rm_1, rm_2 \in RM : \neg(rmState[rm1] = \text{“}aborted\text{”} \wedge rmState[rm2] = \text{“}committed\text{”})$$

$$Inv23 \triangleq \forall rm \in RM : (\text{“}Commit\text{”} \in msgsCommit) \Rightarrow (rmState[rm] \neq \text{“}aborted\text{”})$$

$$Inv11 \triangleq \forall rm \in RM : \neg(\text{“}Abort\text{”} \in msgsAbort) \vee (rmState[rm] \neq \text{“}committed\text{”})$$

$$Inv7 \triangleq \neg(\text{“}Abort\text{”} \in msgsAbort) \vee (\neg(tmState = \text{“}init\text{”}))$$

Figure 7: Definitions of some lemma nodes from proof graph for *TwoPhase* in Figure 6.

establishes the core safety property of two-phase commit which states that no two resource managers can come to conflicting commit and abort decisions, and provides an intuitive way to understand the structure of this inductive proof.

As seen in Figure 6, the root safety node, *Safety*, has 3 supporting action nodes, *RMRcvCommitMsg*, *RMRcvAbortMsg* and *RMChooseToAbort*, which represent, respectively, the actions that can directly falsify the target safety property of two-phase commit, via some resource manager committing or aborting. The support lemmas of *RMRcvCommitMsg* and *RMRcvAbortMsg* respectively, *Inv*23 and *Inv*11, for example, stipulate that presence of a commit or abort message must imply that no other resource manager has made a conflicting commit (abort) decision (definitions shown in Figure 7). This reasoning can also be seen tracing back to lower level support lemmas in the graph e.g., *Inv*7 which establishes invariants on the initial state of the transaction manager.

Overall, this proof graph admits a relatively tree-like structure, and we can naturally focus on small sub-components on the graph. We found that this case study generally supports our interpretability hypothesis i.e., that the inductive proof graph structure can provide insight into the reasoning structure of the proof, and can help guide its development and analysis.

5.1 Detailed Case Study: Raft Consensus Protocol

As a more in-depth evaluation of our technique for verifying a large scale protocol, we developed an inductive proof for a large-scale, asynchronous specification of Raft, to explore how our automated techniques and interpretability features are effective at facilitating this process. We verified a high level lemma of Raft, *PrimaryOwnsEntries*, which states that if a log entry in Raft exists in term T, then a leader in term T must have this entry in its own log.

Figure 8 shows the complete inductive proof graph that we synthesized for establishing this property of our Raft spec-

ification, which consists of 37 lemma nodes. The main actions of this proof graph correspond to those of standard Raft e.g., dealing with election of a leader (*RequestVote*, *HandleReqVoteReq*, *HandleReqVoteResp*, *BecomeLeader*) and replicating log entries between servers (*AppendEntries*, *AcceptAppendEntries*, *ClientRequest*). The core safety property is shown in green, and we show several important core synthesized support lemmas annotated in blue, which serve as helpful waypoints for understanding the structural components of this proof graph.

Specifically, along the *ClientRequest* support ancestry of the safety node, this leads to a first main support lemma, *ElectionSafety*, which is a key property of standard Raft stating that two leaders cannot be elected in the same term. Separately, the *BecomeLeader* parent subgraph of the safety property is supported by the *CandidateElectImpliesNoLogsInTerm* lemma, stating that if a candidate has gathered a quorum of votes, then there mustn't exist any logs in its term. The supporting subgraph for the *ElectionSafety* property, highlighted in orange, is largely independent of the subgraph supporting *CandidateElectImpliesNoLogsInTerm*, highlighting the compositional substructure of this graph.

Another key feature made apparent in this graph is where so-called *message induction cycles* occur. That is, cases where certain lemmas must hold both on a local server and also when its state is sent into the network via a message. For example, *Safety* and *Inv7970* form one of these cycles, where *Inv7970* states a similar property to *PrimaryOwnsEntries* but refers to the log state in an *AppendEntries* message rather than the state of a local server. Similarly, *CandidateElectImpliesNoLogsInTerm* and *Inv13260* form such a cycle. 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 [50], and some invariants can be automatically derived from others.

In general, we found the proof graph structure a significant aid to developing this inductive proof with both automation and guidance. During development, we encountered several cases where our synthesis algorithm was able to discharge significant portions of the graph but failed on key local nodes. We found a highly localized manner of inspection and grammar repair to be very effective in ultimately guiding the tool towards full convergence. Without the automation and interpretability features of our technique, we do not think it would have been possible to efficiently easily 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 [13], SWISS [17] DuoAI [47, 48], and others [38]. 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 [11] and Daikon [8] both use enumerative checking approaches to discover program invariants. FreqHorn [10] 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. Other techniques have also tried to make invariant discovery more efficient by using improved search strategies based on MCMC sampling [39].

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 [35] and additional related work on exploiting modularity for decidability [42].

In the Ivy system [35] 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 [21]. 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.

More generally, compositional verification has a long history and has been employed as a key technique for addressing complexity of large scale systems verification. For example, previous work has tried to decompose proofs into decidable sub-problems [42]. The notion of learning assumptions for compositional assume-guarantee reasoning has also been explored thoroughly and bears similarities to our approach of learning support lemmas while working backwards from a target proof goal [5]. Compositional model checking techniques have also been explored in various other domains [2, 30].

Concurrent Program Analysis Our techniques presented in this paper bear similarities to prior approaches used in

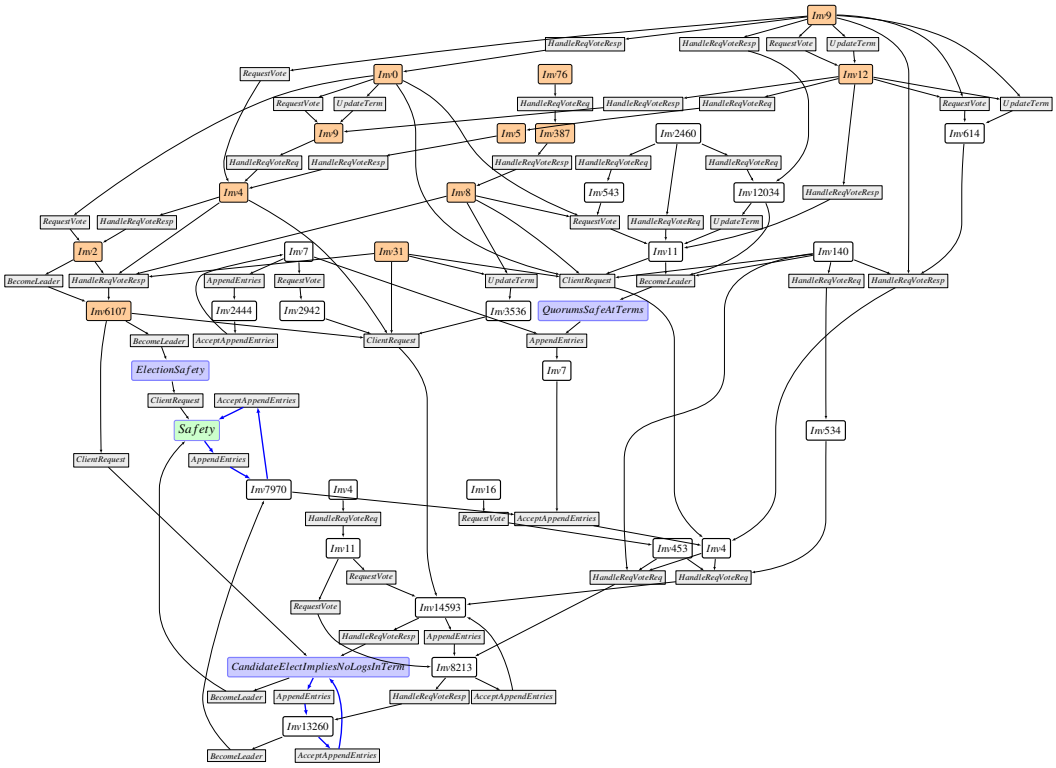


Figure 8: *AsyncRaft* inductive proof graph for *PrimaryOwnsEntries* safety property (labeled as *Safety* in green). Key lemmas named and shown in blue. Key lemmas in blue, support subgraph for *ElectionSafety* in orange, and induction cycles in blue.

the analysis and proofs of concurrent programs. Our notion of inductive proof graphs is similar to the *inductive data flow graph* concept presented in [9]. 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 [15], as well as other *program slicing* techniques [43]. It also shares some concepts with other path-based program analysis techniques that incorporate slicing techniques [19, 20]. 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. For example, we would be interested in seeing how the compositional structure of the inductive proof graph can be used to further tune and optimize local inference tasks e.g. by taking advantage of more local properties that can accelerate inference, like specialized quantifier prefix templates, action-specific grammars, etc. 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.

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., Henzinger, T. A., Mang, F. Y. C., Qadeer, S., Rajamani, S. K., and Tasiran, S. Mocha: Modularity in model checking. In *Computer Aided Verification* (Berlin, Heidelberg, 1998), A. J. Hu

and M. Y. Vardi, Eds., Springer Berlin Heidelberg, pp. 521–525.

[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] CHOU, C.-T., MANNAVA, P. K., AND PARK, S. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings 5* (2004), Springer, pp. 382–398.

[5] COBLEIGH, J. M., GIANNAKOPOULOU, D., AND PĂSĂREANU, C. S. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems: 9th International Conference, TACAS 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 9* (2003), Springer, pp. 331–346.

[6] 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).

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] GOEL, A., MERZ, S., AND SAKALLAH, K. A. Towards an automatic proof of the bakery algorithm. In *Formal Techniques for Distributed Objects, Components, and Systems* (Cham, 2023), M. Huisman and A. Ravara, Eds., Springer Nature Switzerland, pp. 21–28.

[13] 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.

[14] 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.

[15] 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.

[16] GRAY, J. Notes on data base operating systems. In *Operating Systems, An Advanced Course* (Berlin, Heidelberg, 1978), Springer-Verlag, p. 393–481.

[17] 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.

[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] 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.

[20] 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.

[21] 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).

[22] 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.

[23] 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.

[24] LAMPORT, L. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM 17*, 8 (aug 1974), 453–455.

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

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

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

[28] 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).

[29] MANNA, Z., AND PNUELI, A. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, Heidelberg, 1995.

[30] MCMILLAN, K. A methodology for hardware verification using compositional model checking. *Science of Computer Programming 37*, 1 (2000), 279–309.

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

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

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] SHARMA, R., AND AIKEN, A. From Invariant Checking to Invariant Inference Using Randomized Search. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings* (2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 88–105.

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

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.