Interactive Safety Verification of Distributed Protocols by Inductive Proof Decomposition

William Schultz[1], Edward Ashton[2], Heidi Howard[2], and Stavros Tripakis[1]

[1] Northeastern University, Boston, USA
[2] Microsoft Research, Cambridge, UK

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

1 Introduction

Verifying the safety of large-scale distributed and concurrent systems remains an important and difficult challenge. These protocols serve as the foundation of many modern fault-tolerant systems, making the correctness of these protocols critical to the reliability of large scale database and cloud systems [40,15,5]. 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 [9,48,36,21]. Many of these tools are based on modern model checking algorithms like IC3/PDR [20,21,9,10,19], and others based on syntax-guided or enumerative invariant synthesis methods [12,47]. 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 [25,10]. The theoretical complexity limits facing these techniques, however, limit their ability to be fully general [35] and, even in practice, the performance of these tools on complex protocols is still unpredictable, and their failure modes can be opaque. In particular, a key drawback of these methods is that, in their current form, they are very much "all or nothing". That is, if they solve a given problem, no manual proof effort is needed, but if a problem falls outside the scope of what they can solve, little assistance is provided in terms of how to develop a manual proof or how a human can offer guidance to the tool.

In practice, real world, large-scale verification efforts often require some amount of human interaction i.e., a human provides guidance when an automated engine is unable to automatically prove certain properties about a design or protocol. For example, recent verification efforts of industrial scale protocols either note the high amount of human effort in developing inductive invariants or leave them as future goals [39,3]. Several recent, automated approaches have also adopted a paradigm of integrating human assistance to accelerate proofs for larger verification problems e.g., in the form of a manually developed refinement hierarchy [10,29].

Though there has been a large amount of work on scaling *automated* protocol verification techniques, there has been considerably less focus on *interactive* verification. That is, consideration of how a human can proceed effectively with an inductive proof when a tool fails to solve a verification task automatically. The Ivy framework [36] was a notable, more recent attempt to address the interactive safety verification problem, but its techniques were targeted at specific goals which only addressed partial aspects of the problem. Namely, their focus was primarily on (1) ensuring decidability of verification conditions and (2) incorporating a human into the loop of counterexample generalization heuristics. Though this addressed some aspects of the human-machine verification interface, it did not consider other, key issues that arise in large-scale inductive proof efforts. For example, it did not consider how to manage the structure of a large inductive invariant effectively as it is being developed, how to provide feedback to a user about progress on the proof, or how to effectively allow localized reasoning on sub-components of a larger proof, etc.

In addition to frameworks like Ivy, there is a large amount of work on the use of interactive theorem proving for system verification [13,4] e.g., using systems like Coq [1], Isabelle/HOL [32], ACL2 [16], etc. The learning curve for these tools is typically steep, though, and they have typically offered a significantly lower degree of automation, making them more laborious to use for many verification efforts and for protocol designers or engineers [31]. Thus, although these tools provide a relatively high degree of interactivity, they are often quite complex and tedious to use for practical verification efforts.

In this paper we present a new, interactive safety verification methodology, *inductive proof decomposition*, that provides a compositional approach to interactive inductive invariant development, enabling a smooth integration between human effort and machine guidance for large-scale safety proof efforts. We are focused on the human-aided development of inductive invariants, typically with the assistance of a backend solver for checking inductive proof obligations that can provide counterexamples to induction. Standard approaches to this process (e.g., as in the paradigm of Ivy) essentially proceed by having a human verifier examine induction counterexamples in a linear fashion, with a goal of constructing a monolithic list of lemma invariants whose conjunction form a valid inductive invariant. We argue that this standard model is poorly suited for large and complex verification efforts, where inductive invariants may grow to include potentially dozens of complex conjuncts about protocol state and individual counterexamples become increasingly complex to analyze. Our technique aims to make this process fundamentally *compositional*, which we believe is a core aspect of any "intuitive" proof process undertaken by a human (e.g. as in "pen and paper" style proofs), and is also crucial for managing complexity in any large proof effort [24,44,41].

Our technique is based around a novel, formal structure which we introduce and define, the *inductive proof graph*, which imposes a particular compositional structure on an inductive invariant, while also taking into account the distinct logical actions that are present in most concurrent and distributed protocols. This graph structure makes explicit the relative induction dependencies between lemmas of a monolithic inductive invariant, and our proof methodology guides a human verifier to incrementally construct this graph structure by working backwards from a target safety property. Throughout the process, machine guidance is provided in the form of relative induction counterexamples that are maintained at each node of this graph, and so can be reasoned about locally, rather than considered in the scope of the entire inductive invariant. This also allows for local abstractions to be applied that reduce the complexity of counterexamples to be examined. In particular, our approach includes a novel *variable slicing* technique, that projects out protocol variables that are irrelevant to a local node of the proof graph, often significantly reducing the scope of information presented to a user, easing their analysis task.

We apply our technique to several distributed protocols, including a large, industrial-scale specification of the Raft [34] consensus protocol, demonstrating the effectiveness of our technique and its ability to allow human verifiers to work with large proof structures effectively. We also show how the resulting proof graph artifacts provide additional insight into protocol correctness.

In summary, our contributions are as follows:

– Definition and formalization of *inductive proof graphs*, a formal structure representing the logical dependencies between conjuncts of an inductive invariant and actions of a distributed or concurrent protocol. (Section 3)

– *Inductive proof decomposition*, a methodology for large scale interactive safety proofs that is based around the incremental, counterexample-guided construction of an inductive proof graph. (Section 4)
– Implementation of our technique in an interactive verification tool, Scimitar, and evaluation of our method on several distributed protocols, including a large-scale specification of the Raft [34] protocol. (Section 5)

2 Preliminaries

In this paper we are focused on the problem of safety verification of protocols formalized as discrete transition systems, which consists of a core problem of finding adequate inductive invariants. Furthermore, we are focused on verification of systems that are assumed to be correct i.e., we assume various bug-finding methods ([14],[2]) have been applied upfront before a proof is undertaken.

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

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

Inductive Invariants and Relative Induction The standard technique for proving an invariant S of a system $M = (I, T)$ is to develop an *inductive invariant* [30], CONSTANTS $Node, Value, Quorum$

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

Protocol actions.

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

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

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

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

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

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

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

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

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

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

Fig. 1: Summarized specification of *SimpleConsensus*, a simple leader-based consensus protocol. The protocol is parameterized on a finite set of nodes ($Node$) and values ($Value$), and nodes elect one leader to decide on a value. The safety property ($NoConflictingValues$) is an invariant stating that no two nodes can decide conflicting values. The associated inductive invariant, Ind, is used to establish this safety property.

which is a state predicate Ind such that $Ind \Rightarrow S$ and

$$I \Rightarrow Ind \tag{1}$$
$$Ind \land T \Rightarrow Ind' \tag{2}$$

where Ind' denotes the predicate Ind where state variables are replaced by their primed, next-state versions. Conditions (1) and (2) are, respectively, referred to as *initiation* and *consecution*. Condition (1) states that Ind holds at all initial states. Condition (2) states that Ind is *inductive*, i.e., if it holds at some state s then it also holds at any successor of s. Together these two conditions imply that Ind is also an invariant, i.e., that Ind holds at all reachable states.

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

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 localized reasoning and human interpretability, and also to integration of machine assistance, 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 [30]. Any monolithic inductive invariant, however, can alternatively be viewed in terms of a *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

$$(S \wedge L_1 \wedge \cdots \wedge L_k) \wedge T \Rightarrow (S \wedge L_1 \wedge \cdots \wedge L_k)' \tag{3}$$

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

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

If the overall invariant Ind is inductive, then each of the proof obligations in Formula 4 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., $\left(\bigwedge_{\ell \in U} \ell\right) \wedge L \wedge T \Rightarrow L'$. As shown above in Formula 4, $\mathcal{L}$ is always a support set

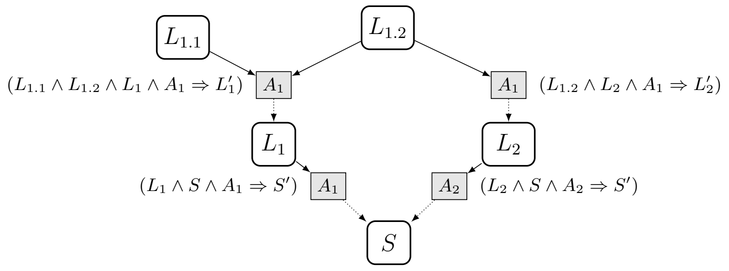

$L_{1.1}$
$L_{1.2}$
$(L_{1.1} \wedge L_{1.2} \wedge L_1 \wedge A_1 \Rightarrow L_1')$
A_1
A_1
$(L_{1.2} \wedge L_2 \wedge A_1 \Rightarrow L_2')$
L_1
L_2
$(L_1 \wedge S \wedge A_1 \Rightarrow S')$
A_1
A_2
$(L_2 \wedge S \wedge A_2 \Rightarrow S')$
S

Fig. 2: Abstract inductive proof graph example, with lemma and action nodes (gray boxes), and associated inductive proof obligations next to each action node. Self-inductive obligations omitted for clarity.

for any lemma in $\mathcal{L}$, but it may not be the smallest support set. This support set notion gives rise to 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 most 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 2 shows an abstract example of an inductive proof graph along with its corresponding inductive proof obligations annotating each action node. For simplicity, when depicting inductive proof graphs, if an action node is self-inductive, we omit it. Similarly, action nodes are always associated with a particular lemma, so we visualize edges connecting action nodes to their parent lemma node, even though these are not edges 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 2 (Local 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:*

$$\left(\wedge_{\ell \in Supp_{(L,A)}} \ell\right) \wedge L \wedge A \Rightarrow L' \tag{5}$$

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

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

Theorem 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, and all lemmas L are invariants of M.*

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. Additionally, for any set of predicates, if their conjunction is an invariant of M, then each conjunct must be an invariant of M, so all such lemmas are necessarily invariants of M.

3.3 Local Variable Slices

A valuable feature of the inductive proof graph is that it enables, at each proof node, focus on a smaller subset of state variables relevant for discharging that 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. This general idea allows for the computation of a *variable slice* at each node, projecting away 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. 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 4. *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 5. *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, 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 2. *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 existence of a predicate $Supp$ such that $Supp \wedge L \wedge A \wedge \neg L'$ 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 this formula 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 $f_i(\mathcal{D}_i)$. Then, 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 $Slice(L, A)$.

4 Interactive Safety Verification

Having formalized the inductive proof graph structure in Section 3, we now describe our interactive proof methodology, *inductive proof decomposition*, which is based around incremental construction of an inductive proof graph, with integration of localized counterexample guidance and slicing. We start by providing some additional context and motivation in Section 4.1, followed by a description of our procedure by way of a motivating example in Section 4.2.

4.1 Context and Motivation

In a standard interactive safety verification paradigm (e.g., in Ivy [36]), the general technique is based on linear, iterative analysis of counterexamples to induction (CTIs). That is, to develop an inductive invariant such as the one shown in Figure 1, one starts with the target safety property (*NoConflicting-Values*) and generates counterexamples to induction, iteratively developing new lemma invariants to rule out these counterexamples until the overall invariant becomes inductive.

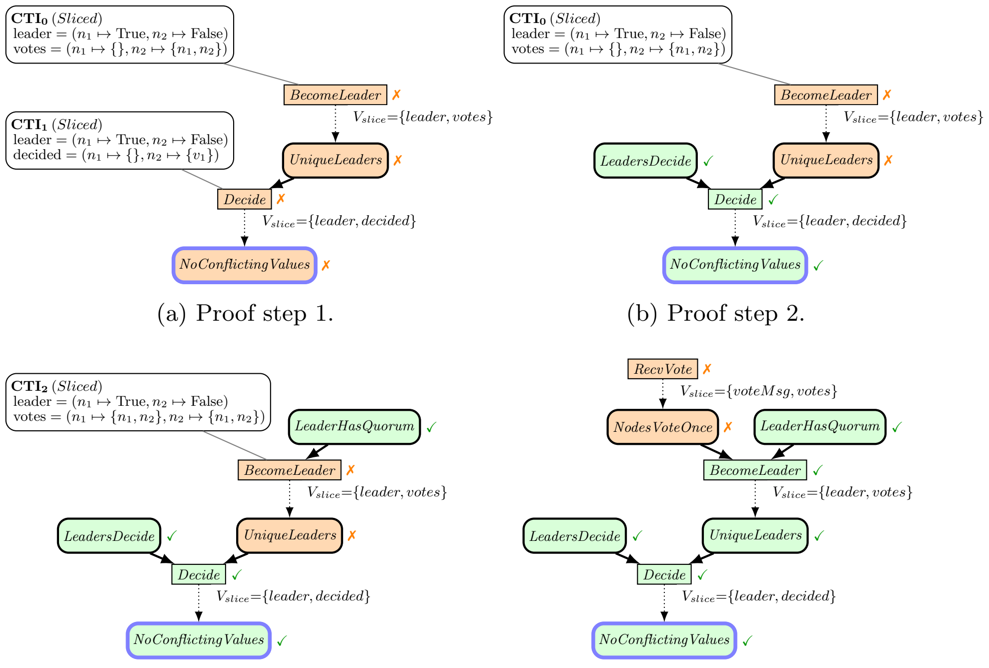
For small protocols and invariants, this basic procedure may be sufficient, but, for large scale verification efforts, its effectiveness breaks down. That is, when systems and their invariants become large, it becomes increasingly difficult to manage and understand the global structure of these inductive invariants as they are being developed, since the interaction between existing lemmas of the invariant candidate and the actions of the system become complex. More concretely, we characterize the issues with existing approaches into the following core themes:

P1. **CTI Management**: How does one manage the set of CTIs for the current inductive invariant and decide which CTI from this set to analyze?
P2. **Localization**: Once a CTI is selected, how does one focus only on the lemmas, actions, and state variables relevant to analysis of this CTI?
P3. **Proof Status and Structure**: As new lemmas are developed, how can the current proof status, structure, and progress be measured?

Our technique, which we describe in the following section, is largely motivated by the fact that, to our knowledge, no existing proof methodologies provide a formal, conceptual framework for addressing the above questions. Thus, existing counterexample-guided inductive invariant development processes are opaque and can be extremely laborious even for relatively experienced protocol designers. In the following section we explain the core ideas and description of our technique and how it aims to address these issues.

4.2 Interactive Verification Procedure

We illustrate our interactive verification technique by using it to work through the partial development of an inductive invariant for the *SimpleConsensus* protocol, defined as a symbolic transition system in Figure 1. This protocol utilizes a simple leader election scheme to select values, and is parameterized on a set of nodes, *Node*, a set of values to be chosen, *Value*, and *Quorum*, a set of intersecting subsets of *Node*. Nodes can vote once for another node to become leader, and once a node s garners a quorum of votes it may become leader and decide a value by setting its local $decided[s]$ variable. The target safety property, *NoConflictingValues*, shown in Figure 1, states that no two differing values can be decided upon by different nodes. Also shown there is a complete, monolithic inductive invariant, *Ind*, for establishing this safety property, consisting of 8 lemma conjuncts, along with a subset of definitions for the lemmas in *Ind*.


CTI₀ (Sliced)
leader = (n₁ ↦ True, n₂ ↦ False)
votes = (n₁ ↦ {}, n₂ ↦ {n₁, n₂})
BecomeLeader ✗
V_slice={leader, votes}
CTI₁ (Sliced)
leader = (n₁ ↦ True, n₂ ↦ False)
decided = (n₁ ↦ {}, n₂ ↦ {v₁})
UniqueLeaders ✗
Decide ✗
V_slice={leader, decided}
NoConflictingValues ✗
LeadersDecide ✓
CTI₂ (Sliced)
leader = (n₁ ↦ True, n₂ ↦ False)
votes = (n₁ ↦ {n₁, n₂}, n₂ ↦ {n₁, n₂})
LeaderHasQuorum ✓
RecvVote ✗
V_slice={voteMsg, votes}
NodesVoteOnce ✗

(a) Proof step 1. (b) Proof step 2. (c) Proof step 3. (d) Proof step 4.

Fig. 3: Example progression of inductive proof graph development for invariant of Fig. 1. Nodes in orange(✗) are those with remaining inductive proof obligations to be discharged, and those in green(✓) have all obligations discharged. CTI pre-states are shown as annotations associated with relevant action nodes.

Figure 3a shows an in-progress inductive proof graph that corresponds to a partially completed inductive invariant containing the first two conjuncts of Ind from Figure 1. Our invariant development procedure now starts from this inductive proof graph, from which the possible next steps in our process are clear to assess. In particular, it is clear that the *Decide* and *BecomeLeader* nodes are unproven (shown in orange with ✗), meaning that there are outstanding CTIs for the inductive proof obligations of those nodes as explained above. Additionally, we can focus on separate CTIs in isolation, since CTIs are associated with specific action nodes. This makes it clear which lemmas and actions these CTIs are relevant to, alleviating the issues of P1 as described above.

To proceed, we then work to extend this graph by developing appropriate support lemmas and associated edges until nodes are discharged (i.e. made valid). For example, we may first select $\mathbf{CTI_1}$ to analyze, as shown in Figure 3a. In addition to the localization of counterexamples, the decomposition provided by the proof graph also allows for localized state variable slices to be applied to the CTIs at each action node, and are shown as V_{slice} alongside each action node. For example, at the *Decide* node in Figure 3a, which $\mathbf{CTI_1}$ is associated with,

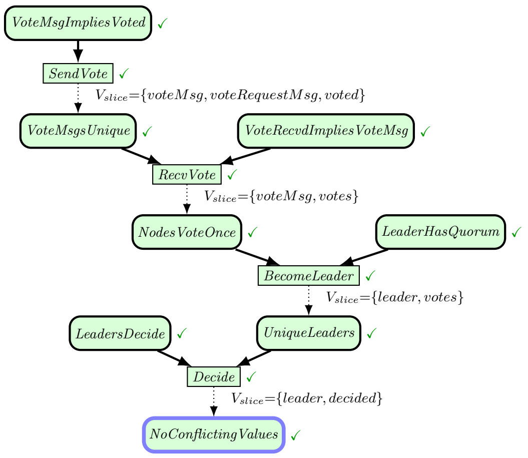


Fig. 4: Complete inductive proof graph for *SimpleConsensus* and safety property *NoConflictingValues*, with variable slice annotations.

$V_{slice} = \{leader, decided\}$, containing only 2 out of 6 total state variables of the *SimpleConsensus* system. These slices, as formalized in Section 3.3, significantly reduce the number of state variables to be considered at each node, greatly mitigating the analysis burden on a human user and addressing the issues raised in P2 above.

After analyzing $\mathbf{CTI_1}$ we may develop the *LeadersDecide* lemma, which states that only leaders could have decided values, and rules out $\mathbf{CTI_1}$. *LeadersDecide* is then added as a new incoming support lemma of the *Decide* action node, leading to the proof graph state shown in Figure 3b. At this point, we see from the graph status that lemmas *NoConflictingValues* and *LeadersDecide* are discharged, so we no longer need to consider these lemmas in our reasoning. This gives us a useful dynamic measure of proof status and logical structure as we develop new lemmas over time, alleviating the issue of P3 above.

Next, as shown in Figure 3b, we have a remaining counterexample, $\mathbf{CTI_0}$, associated with the unproven *BecomeLeader* node. Here, we again benefit from local slicing, which gives $V_{slice} = \{leader, votes\}$ at this *BecomeLeader* node. Analysis of $\mathbf{CTI_0}$ yields the *LeaderHasQuorum* support lemma, which is self-inductive, leading us to the proof state in Figure 3c. From there, we develop one additional support lemma to rule out $\mathbf{CTI_2}$, giving us the *NodesVoteOnce* support lemma, which is sufficient to discharge the *UniqueLeaders* lemma node, leading us to the proof state shown in Figure 3d. We do not show a further progression of the proof process for this graph, but we can see from Figure 3d that this process can be continued in a backwards fashion, starting now from the remaining unproven node *NodesVoteOnce* and its associated action node $RecvVote$. A fully completed proof graph is shown in Figure 4, where every node has been discharged i.e., has a valid set of support lemmas.

Procedure 1 Abstract steps of our interactive verification procedure.

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

Even on this relatively small protocol and invariant, this example demonstrates the value of our interactive, compositional proof methodology. Our decomposition based on the inductive proof graph structure makes explicit the relationship between lemmas and protocol actions as the inductive invariant is being built, and, enabled by this structure, we are able to localize the analysis of CTIs to specific nodes of this graph, along with the benefits of local slicing. We describe this general procedure slightly more formally in Procedure 1. That is, our high level interactive safety verification procedure, for proving that S is an invariant of transition system $M = (I, T)$ by finding an inductive invariant Ind, centers around the incremental construction of an inductive proof graph, working backwards from S as the target invariant.

5 Empirical Evaluation

To evaluate our technique, we used it to develop inductive invariants for several distributed protocols of varying complexity, including development of an inductive invariant for a large scale specification of the Raft [34] consensus protocol, the first inductive invariant effort for a Raft protocol specification of this complexity with this degree of automation.

An artifact containing all of our source code and instructions for reproducing our evaluation results can be found at [37]. A public, open-source version of our tool is also available at [38].

Implementation and Benchmarks We implemented our inductive proof decomposition technique in a tool, Scimitar, that provides a graphical user interface for the interactive development of inductive proof graphs. Scimitar is implemented in Python and accepts systems specified in the TLA$^+$ specification language [26].

The tool allows a user to view a current inductive proof graph, with visualizations representing the state of each lemma and action node. A user can then choose to focus on a particular action node, and a subset of CTIs associated with that node are presented. Slicing is applied automatically to the presented CTIs based on an automatic static analysis of the protocol specification, and the interface also provides a way to for a user to dynamically add support edges and new lemmas to the graph.

Internally, Scimitar uses a combination of the TLC model checker [49] and the Apalache symbolic model checker [22] for checking inductive proof obligations and generating CTIs. Note that TLC is an explicit state model checker, so cannot produce full proofs of inductive invariants for protocols of arbitrary size, but can generate CTIs for finite protocol instances using a randomized search technique [27]. We used finite parameter bounds for counterexample generation with TLC and Apalache, and then also use the TLA$^+$ proof system (TLAPS) [6] to fully validate the final inductive proof graphs. In our experience, TLC can, in some cases, be more efficient than Apalache at generating CTIs, so we found both tools complementary in our implementation. All experiments and proof checking results below were collected using Apalache version 0.44.0 and TLC version 2.15 running on a 2023 M3 Macbook Pro.

We used our tool to develop inductive invariants for establishing core safety properties of 5 distributed protocol specifications. These protocols are all specified in TLA$^+$, and summarized in Table 1, along with various specification and proof statistics. Of the protocols tested, we consider the following 3 to be of medium complexity:

- *SimpleConsensus*: An abstract consensus protocol where nodes elect a leader which then decides on a value, as shown in Figure 1.
- *TwoPhase*: A specification of the two-phase commit protocol [28]. The safety property is *TCConsistent*, stating that two resource managers cannot have conflicting commit or abort decisions.
- *AbstractRaft*: A high level specification of the Raft consensus protocol [34], abstracting away message passing. The safety property is *StateMachineSafety*, stating that committed entries must be consistent across nodes.

The 2 additional protocols are of significantly larger complexity:

- *Bakery*: A specification of Lamport's Bakery algorithm [23] for mutual exclusion. The safety property is *MutualExclusion*, stating two processes cannot be in a critical section simultaneously.
- *AsyncRaft*: An industrial scale specification of the Raft consensus protocol [34], based on [33,43], and which models asynchronous message passing between nodes and fine-grained local state. The safety property is *NoLogDivergence*, which asserts that committed indices across nodes are consistent.

The 2 large protocol specifications are both of a complexity significantly greater than those tested in recent automated invariant inference techniques, so are more relevant for evaluating our interactive verification techniques.

Protocol	Variables	Actions	LOC (spec/pf)	Lemmas	Median Slice Size
SimpleConsensus	6	5	111 / 18	8	2 (0.33)
TwoPhase	6	7	192 / 140	16	3 (0.50)
AbstractRaft	4	6	220 / 138	10	3 (0.75)
Bakery	6	14	283 / 116	27	4 (0.67)
AsyncRaft	12	11	638 / 920	39	5 (0.42)

Table 1: Protocol benchmarks and associated metrics. *LOC* is the number of lines of TLA$^+$ code for defining the specification (spec) and inductive proof (pf), respectively. *Median slice* gives the median size of all variable slices in the complete inductive proof graph, and its proportion of the total state variables.

5.1 Results and Discussion

Table 1 shows various statistics about the protocols we tested, including the number of state variables, number of actions, lines of code (LOC) in the TLA$^+$ protocol specifications and proofs, number of lemmas in each proof graph, and the size of variable slices. We discuss the structure of the developed proof graphs and qualitative aspects of our experience developing these proofs in more detail below.

Medium Protocols Visualizations of the completed inductive proof graphs for *AbstractRaft* and *TwoPhase*, respectively, can be seen in Figures 5 and 6, and the proof graph for *SimpleConsensus* was shown previously, in Figure 4. The full graphs can be viewed and explored using our interactive tool.

Though these proof graphs are for smaller protocols, they generally confirm our intuition that these graphs are useful for understanding the structure of an inductive proof and guiding its development. Even for protocols of this size, we found that maintenance of this structure coupled with automatic counterexample slicing is a significant aid, and allowed us to develop these inductive invariants with much more ease and efficiency. In *SimpleConsensus*, for example, the median slice proportion is 0.33, indicating that most slices present a significantly reduced amount of information, and the graph admits a clear tree-like decomposition. Our experience for *TwoPhase* was similar, also benefitting from slicing with a median slice proportion of 0.5.

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

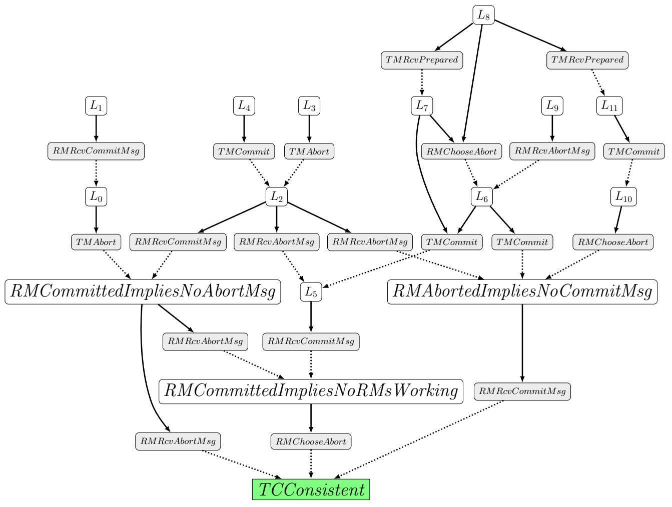


Fig. 5: *TwoPhase* inductive proof graph.

Large Protocols As an exploration of our technique for larger scale proof efforts, we applied it first to *AsyncRaft*, a specification of Raft at a considerably lower level of abstraction e.g., it includes asynchronous message passing and fine-grained local state, akin to the detail level provided in Raft's original TLA$^+$ specification [33]. We are also aware of no prior inductive invariant that existed for a TLA$^+$ specification of Raft at this level of complexity. Figure 7 shows a visualization of our completed inductive proof graph for *AsyncRaft*.

Overall, we found our technique effective at making the proof process efficient and understandable for a human verifier, by providing a formal structure to the large scale proof and accelerating CTI analysis by slicing and localized reasoning. We were able to develop such an inductive invariant in approximately 3 human weeks of effort, which we found to be a productive pace for an invariant of this size and protocol of this complexity. Other verification efforts of this type note, for example, an inductive invariant development burden of 1-2 human months [39], even for a protocol with a smaller invariant than *AsyncRaft*. We found that the decomposition provided by our technique enabled effective local reasoning in many cases. For example, comparing the slices near *LeaderMatchIndexValid* and *LeaderMatchIndexBound* nodes to *QuorumsSafeAtTerms*, in a relatively distinct sub-component of the graph, we can see that the slice sets refer to relatively disjoint subsets of variables (out of 12 total variables), supporting our hypothesis that the compositional structure of the proof graph allows for localized reasoning on different aspects of the protocol.

In addition to the efficiency of invariant development, we also found value in the developed proof artifact beyond establishing correctness. For example, during the development of the proof, we found it helpful to observe various patterns that arise in the proof graph structure. In particular, we observed the various

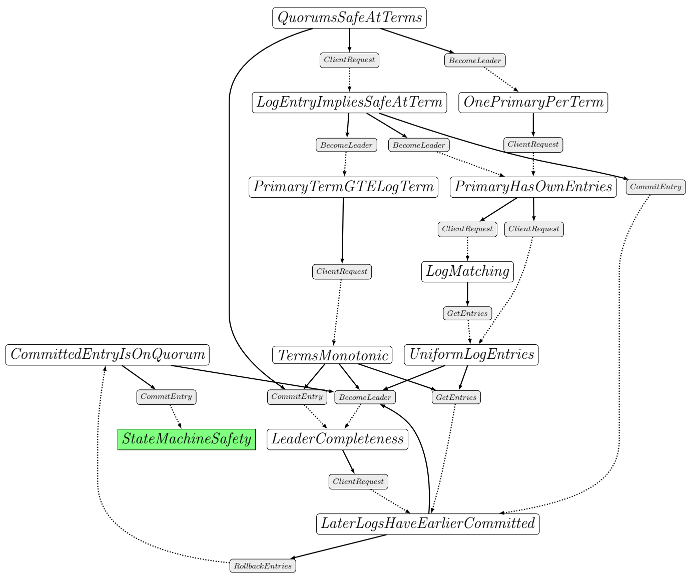


Fig. 6: *AbstractRaft* inductive proof graph.

types of *induction cycles* that arise in this graph, especially those that arise in many places due to the message passing nature of this protocol. For example, we can observe the cycle highlighted in Figure 7 where *LogMatching* serves as a support lemma of *LogMatchingInAppendEntriesMsgs* via the *AppendEntries* action, and *LogMatchingInAppendEntriesMsgs* then serves as a support lemma of *LogMatching* via the *AppendEntriesResponse* action, forming this induction cycle. These cycles would be difficult to observe in a standard inductive invariant, but are apparent in our proof graph structure, and represent a common pattern where invariants about protocol state must hold on both local state and also on the state of messages that are sent over the network. We found a similar experience with *Bakery* proof development, but omit its full proof graph for sake of space, showing its summarized proof statistics in Table 1.

6 Related Work

There are two recent works that are most similar to ours in scope and approach, namely, the Ivy system [36] and the work on exploiting modularity for decidability presented in [41]. Our approach bears similarities to these other two verification techniques, but there are a number of key differences in terms of goals and methodologies.

One main focus of Ivy is on the modeling language, with a goal of making it easy to represent systems in a decidable fragment of first order logic, so as to ensure verification conditions always provide some concrete feedback in the

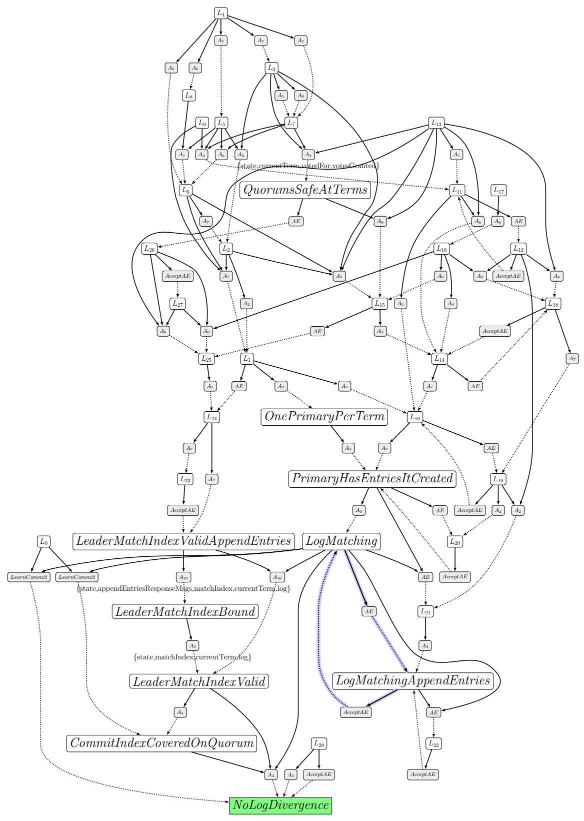


Fig. 7: *AsyncRaft* inductive proof graph structure, with excerpted action nodes and lemmas shown with detailed names, and induction cycle highlighted in blue.

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 [19]. In contrast, our work is primarily focused on different concerns e.g., we focus on compositionality as a means to provide an efficient and scalable proof methodology, and as a means to produce a more interpretable proof artifact, in addition to allowing for localized counterexample reasoning. We also view decidable modeling as an orthogonal component of the verification process that could be complementary to our approach.

The goals of the work in [41] are more similar to our own, in that they aim to use a particular type of compositional reasoning to exploit decidable subproblems when possible. This is perhaps closest to our approach in that it tries to exploit decomposition in the verification and proof process, but the decomposition notions and the way they are used are somewhat different between our work and theirs. Our goal is to define a compositional structure that is integrated into the counterexample guidance process, while also producing a single, inductive proof artifact upon completion. In future it would be interesting, however, to consider whether our notion of decomposition based on the inductive proof graph structure admits a similar kind of "decidable subproblem" property that is exploited in [41].

Our techniques also bear similarities to prior approaches for the proofs and analysis of concurrent programs, namely that of *inductive data flow graphs* [8]. That work, however, is focused on the verification of multi-process concurrent programs, and did not generalize the notions to a distributed setting. The procedures for verification and counterexample analysis are also different between our approach and theirs. Our compositional approach also bears similarity to recent work on *recomposition* techniques for verification [7]. Generally, our slicing technique is similar to a cone-of-influence reduction [11], as well as other *program slicing* techniques [42]. It also shares some concepts with other path-based program analysis techniques that incorporate slicing techniques [17,18]. In our case, however, we apply it at the level of a single protocol action and target lemma, particularly for the purpose of CTI state projection.

With respect to fully automated invariant inference, there are several recently published techniques that attempt to solve this problem for distributed protocols, including IC3PO [9], SWISS [12] and DuoAI [47]. 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 modern tools in this area can solve.

7 Conclusions and Future Work

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

References

1. Bertot, Y., and Castéran, P. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media, 2013.
2. Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. Bounded model checking. *Adv. Comput. 58* (2003), 117–148.
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).

4. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang. 1*, ICFP (aug 2017).
5. 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).
6. 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.
7. Dardik, I., Porter, A., and Kang, E. Recomposition: A new technique for efficient compositional verification. In *2024 Formal Methods in Computer-Aided Design (FMCAD)* (2024), pp. 130–141.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, Association for Computing Machinery, p. 1–17.
14. Holzmann, G. J. The model checker SPIN. *IEEE Transactions on software engineering 23*, 5 (1997), 279–295.
15. 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.
16. Hunt Jr, W. A., Kaufmann, M., Moore, J. S., and Slobodova, A. Industrial hardware and software verification with acl2. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 375*, 2104 (2017), 20150399.

17. 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.
18. 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.
19. 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).
20. 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.
21. 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.
22. Konnov, I., Kukovec, J., and Tran, T.-H. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct 2019).
23. Lamport, L. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM 17*, 8 (aug 1974), 453–455.
24. Lamport, L. How to write a proof. *The American mathematical monthly 102*, 7 (1995), 600–608.
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. Lampson, B., and Sturgis, H. Crash Recovery in a Distributed Data Storage System. *Unpublished technical report, Xerox Palo Alto Research Center* (06 1979).
29. 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).
30. Manna, Z., and Pnueli, A. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, Berlin, Heidelberg, 1995.
31. Newcombe, C. Why Amazon Chose TLA+. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z* (2014), Springer, pp. 25–39.
32. Nipkow, T., Wenzel, M., and Paulson, L. C. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer, 2002.
33. Ongaro, D. Consensus: Bridging Theory and Practice. *Doctoral thesis* (2014).
34. 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.
35. 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.

36. 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.
37. Schultz, W. Artifact for NFM 2026 paper: Interactive Safety Verification of Distributed Protocols by Inductive Proof Decomposition. `https://github.com/will62794/nfm26-artifact`, 2026. Accessed: 2026-03-24.
38. Schultz, W. Scimitar: verification tool for distributed protocols based on inductive proof decomposition. `https://github.com/will62794/scimitar`, 2026. Accessed: 2026-03-24.
39. 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.
40. 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.
41. 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.
42. Tip, F. A survey of program slicing techniques. *J. Program. Lang. 3* (1994).
43. 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.
44. Wilcox, J. R. *Compositional and Automated Verification of Distributed Systems*. PhD thesis, University of Washington, USA, 2021.
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.