Scalable, Interpretable Distributed Protocol Verification by Inductive Proof Slicing

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

Many techniques for automated inference of inductive invariants for distributed protocols have been developed over the past several years, but their performance can still be unpredictable and their failure modes opaque for large-scale verification tasks. In this paper, we present *inductive proof slicing*, a new automated, compositional technique for inductive invariant inference that scales effectively to large distributed protocol verification tasks. Our technique is built on a core, novel data structure, the *inductive proof graph*, which explicitly represents the lemma and action dependencies of an inductive invariant and is built incrementally during the inference procedure, backwards from a target safety property. We present an invariant inference algorithm that integrates localized syntax-guided lemma synthesis routines at nodes of this graph, which are accelerated by computation of localized grammar and state variable slices. Additionally, in the case of failure to produce a complete inductive invariant, maintenance of this proof graph structure allows failures to be localized to small sub-components of this graph, enabling fine-grained failure diagnosis and repair by a user. We evaluate our technique on several complex distributed and concurrent protocols, including a large scale specification of the Raft consensus protocol, which is beyond the capabilities of modern distributed protocol verification tools, and also demonstrate how its interpretability features allow effective diagnosis and repair in cases of initial failure.

ACM Reference Format:
William Schultz, Edward Ashton, Heidi Howard, and Stavros Tripakis. 2024. Scalable, Interpretable Distributed Protocol Verification by Inductive Proof Slicing. 1, 1 (April 2024), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [6, 19, 41]. 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].

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

ACM XXXX-XXXX/2024/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [12, 26, 37, 48]. Many of these tools are based on modern model checking algorithms like IC3/PDR [12, 13, 22, 25, 26], and others based on syntax-guided or enumerative invariant synthesis methods [17, 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 [13, 28]. The theoretical complexity limits facing these techniques, however, limit their ability to be fully general [36] 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 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 [4, 38], 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 [13, 31].

In this paper we present a new technique for automated inference of inductive invariants for distributed protocols that aims to improve on both *scalability* and *interpretability* of existing approaches. Our technique, *inductive proof slicing*, utilizes the underlying compositional structure of an inductive invariant to guide and accelerate inference. Our algorithm integrates this compositional structure with a syntax-guided invariant synthesis approach, executing local synthesis tasks on projections of the protocol to incrementally construct an overall inductive invariant. By localizing synthesis tasks to typically very small projections of the full state space and grammar, we make global inference significantly more efficient, requiring inference sub-routines to consider only small, local proof obligations. These slicing features enable our technique to scale to protocols which are too complex for existing tools to solve fully automatically.

Our inference algorithm is built around a core data structure, the *inductive proof graph*, which we introduce and formalize. It defines a compositional structure on the lemma conjuncts of an inductive invariant, while also incorporating the logical actions of a concurrent or distributed protocol. This graph structure makes explicit the induction dependencies between lemmas of an inductive invariant, and serves as a core guidance mechanism for our inference algorithm. The decomposition provided by this graph can also be presented to and interpreted directly by a human user. In particular, failures of synthesis tasks during inference can be localized to particular nodes and grammar slices. This facilitates a concrete and effective diagnosis and repair process, enhancing interpretability of both the final inductive proof and the intermediate results.

We apply our technique to inferring inductive invariants of several large-scale distributed protocol specifications, including large, industrial-scale protocol specifications of the Raft [35] consensus protocol, demonstrating the effectiveness of our technique. We also provide an empirical evaluation of how the interpretability of our method allows for effective diagnosis and repair in cases where full convergence is not initially achieved.

In summary, our contributions are as follows:

CONSTANTS $Node, Value, Quorum$

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

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

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

Protocol actions.

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

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

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

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

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

Safety property.

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

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

- Definition and formalization of *inductive proof graphs*, a formal structure representing the logical dependencies between conjuncts of an inductive invariant and actions of a distributed or concurrent protocol. (Section 4)
- *Inductive proof slicing*, a new compositional, automated inductive invariant inference technique that is scalable to large protocols and provides localized, interpretable diagnosis of inference failures. (Section 5)
- Implementation of our technique in a verification tool and an empirical evaluation on several distributed protocols, including large-scale specifications of the Raft [35] consensus protocol. (Section 6)

2 OVERVIEW

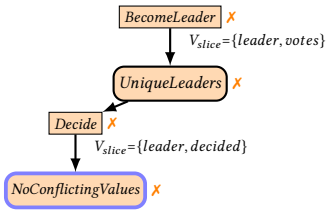
In this section we present a high level overview of *inductive proof slicing*, our automated inductive invariant inference technique. We present the core ideas of our approach below and walk through an example of our inference algorithm on a running example.

Running Example: SimpleConsensus. 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

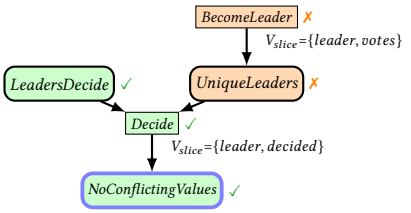
$$\begin{aligned}
&\textit{UniqueLeaders} \triangleq \\
&\quad \forall n_1, n_2 \in \textit{Node} : \ \textit{leader}[n1] \land \textit{leader}[n_2] \implies (n_1 = n_2) \\
&\textit{LeaderHasQuorum} \triangleq \\
&\quad \forall n \in \textit{Node} : \textit{leader}[n] \implies \\
&\quad (\exists Q \in \textit{Quorum} : \textit{votes}[n] = Q) \\
&\textit{LeadersDecide} \triangleq \\
&\quad \forall n \in \textit{Node} : (\textit{decided}[n] \neq \{\}) \implies \textit{leader}[n] \\
&\textit{NodesVoteOnce} \triangleq \\
&\quad \forall n, n_i, n_j \in \textit{Node} : \\
&\neg(n_i \neq n_j \land n \in \textit{votes}[n_i] \land n \in \textit{votes}[n_j])
\end{aligned}$$

$$\begin{aligned}
&\textit{Ind} \triangleq \text{Inductive invariant.} \\
&\quad \land \textit{NoConflictingValues} \ \text{(Safety)} \\
&\quad \land \textit{UniqueLeaders} \\
&\quad \land \textit{LeaderHasQuorum} \\
&\quad \land \textit{LeadersDecide} \\
&\quad \land \textit{NodesVoteOnce} \\
&\quad \land \textit{VoteRecordedImpliesVoteMsg} \\
&\quad \land \textit{VoteMsgsUnique} \\
&\quad \land \textit{VoteMsgImpliesNodeVoted}
\end{aligned}$$

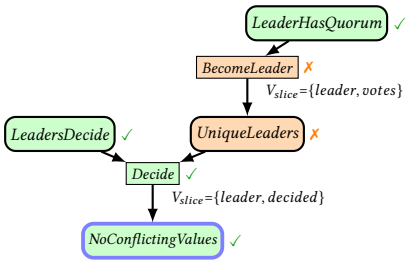
Fig. 2. Complete inductive invariant, *Ind*, for proving the *NoConflictingValues* safety property of the *SimpleConsensus* protocol from Figure 1. Selected lemma definitions from *Ind* are also shown.



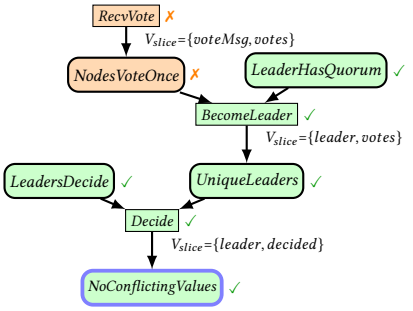
(a) In-progress inductive proof graph for *SimpleConsensus*.



(b) Step 2.



(c) Step 3.



(d) Step 4.

Fig. 3. Example progression of inductive proof graph for *SimpleConsensus* during execution of our inference algorithm. 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.

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.

2.1 Our Approach: Inductive Proof Slicing

Our technique for automated inductive invariant inference, *inductive proof slicing*, utilizes the compositional structure of an inductive invariant to accelerate our inference procedure, and also

to provide fine-grained, interpretable feedback in the case of failure to produce a complete proof. We walk through our technique on our running example below, showing how it incrementally constructs an inductive invariant for the *SimpleConsensus* protocol.

Safety Verification by Inductive Invariant Inference. Our overall goal is to verify that a given protocol satisfies a specified safety property, which we do by automatically discovering an inductive invariant which implies (i.e., is logically stronger than) the safety property. For example, given Figure 1 as input, our technique automatically generates an inductive invariant such as *Ind* shown in Figure 2. *Ind* is the conjunction of the original safety property, *NoConflictingValues*, plus 7 more *lemmas*, which strengthen this safety property (thus ensuring that *Ind* logically implies *NoConflictingValues*). The definitions of 4 of these lemmas are also shown in Figure 2. As can be seen, even for such a relatively simple protocol, the inductive invariant is non-trivial in both size and logical complexity of its predicates.

Inductive Invariant Inference via Inductive Proof Graphs. In our technique, we synthesize an inductive invariant like the one in Figure 2 *incrementally* and *compositionally* using a data structure called an *inductive proof graph*. Our inference algorithm incrementally constructs this graph, working backwards from a specified safety property. Figure 3 shows some initial steps in constructing the inductive proof graph for the *NoConflictingValues* safety property of *SimpleConsensus*. The complete graph is shown in Figure 4.

The main nodes of the 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 *relative 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*. Namely, the requirement to discover a set of supporting lemmas that make L inductive relative to this support set, with respect to action A. More precisely, each action node A with source lemmas $L_1, ..., L_k$ 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. Formula (1) states that if L holds at the current state, and action A is taken, then L will also hold at the next state, provided all lemmas $L_1, ..., L_k$ also hold at the current state.

Our use of this proof graph structure is illustrated more concretely in Figure 3, which shows some initial steps of our algorithm proceeding to synthesize the complete inductive proof graph of Figure 4, for the *NoConflictingValues* safety property of *SimpleConsensus*. Nodes that are unproven (shown in orange and marked with ✗), means that there are outstanding counterexamples for those inductive proof obligations. At each unproven node, our algorithm performs a local invariant synthesis task, to synthesize support lemmas that make the lemma node inductive relative to this set of support lemmas. For example, in Figure 3a we select the unproven *Decide* node and synthesize an additional support lemma, *LeadersDecide*, to rule out counterexamples at that node that were not already eliminated by the existing support lemma, *UniqueLeaders*. After synthesizing the *LeadersDecide* lemma, this becomes a new support lemma of the *Decide* node, which is then marked as proven (shown in green and marked with ✓), as seen in Figure 3b, since all of its induction counterexamples have been eliminated by its set of support lemmas. Our algorithm continues in this fashion, next selecting the unproven *BecomeLeader* node in Figure 3b, and synthesizing the *LeaaderHasQuorum* support lemma. An additional lemma, *NodesVoteOnce*, is then synthesized to discharge the *UniqueLeaders* node, shown in Figure 3d. As seen there, newly synthesized support

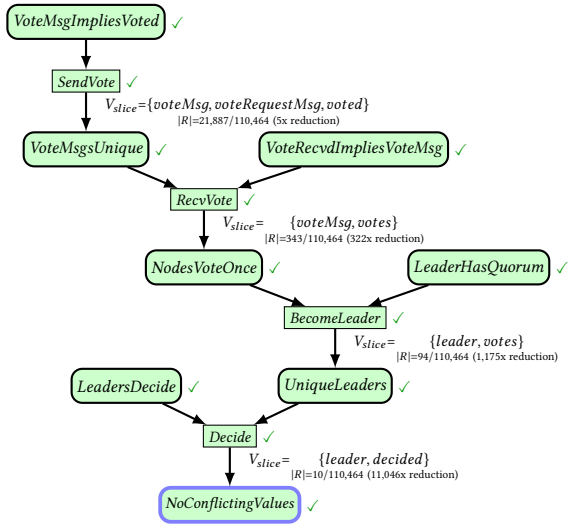


Fig. 4. A complete inductive proof graph for *SimpleConsensus* protocol inductive invariant and *NoConflictingValues* safety property (the root node). Local variable slices are shown as V_{slice}, along with the size of the explored state set slice at that node, indicated as $|R|$, along with the reduction factor over the full set of explored states computed during inference runs of *SimpleConsensus*.

lemmas create new proof obligations to consider (e.g. via *NodesVoteOnce*), and our algorithm will continue until all nodes are discharged e.g., leading to a complete inductive proof graph as shown in Figure 4.

2.1.1 Accelerating Inference with Inductive Proof Slicing. The structure of the inductive proof graph and localized nature of these inference tasks allows for several *slicing* based optimizations, which are key to the performance of our technique. These optimizations are enabled by the computation of a *variable slice* based on the lemma and action pair associated with each action node. This variable slice represents a small subset of state variables which are sufficient to consider when synthesizing support lemmas for a local inductive proof obligation. We use this slice to both prune the search space grammars defining the space of predicates to search over for synthesizing support lemmas (*grammar slicing*) and also to accelerate the tasks of checking candidate invariants for selection as support lemmas (*state slicing*).

For example, in the complete proof graph for *SimpleConsensus* shown in Figure 4, at the *Decide* action node for lemma *NoConflictingValues*, its computed variable slice is $\{leader, decided\}$ (2 out of 6 total state variables). This produces a grammar slice as shown in Figure 5b, a subset of 10/23 total predicates in a full example of a grammar for *SimpleConsensus*, as shown in Figure 5a. We also use this to perform *state slicing*, which projects a cached set of explored system states onto the subset of state variables in this slice. In many cases this yields an order of magnitude reduction in local inference time when searching for candidate invariants, which are validated by checking them on reachable system states. For example, the full set of reachable states explored by our algorithm for the *SimpleConsensus* protocol contains 110,464 distinct states when instantiated with finite parameters where $|Node| = 3$ and $|Value| = 2$. When projected onto the variable slice $\{leader, decided\}$, this set of reachable states contains 10 distinct states, a 11,046x reduction. Similarly, as shown in Figure 4, the variable slice at the *BecomeLeader* action node of *UniqueLeaders* is $\{leader, votes\}$, which gives a projected state set of 94, a 1,175x reduction from the full reachable

$$
\begin{aligned}
&\mathit{Quant} \triangleq \\
&\quad \forall i \in \mathit{Node} : \forall j \in \mathit{Node} : \forall k \in \mathit{Node} : \\
&\quad \exists Q \in \mathit{Quorum} : \forall v \in \mathit{Value} : \\
&\\
&\mathit{Pred} \triangleq \{ \\
&\quad \mathit{voted}[i], \mathit{voted}[j], \mathit{voted}[k], \\
&\quad \langle i, j \rangle \in \mathit{vote\_msg}, \\
&\quad \langle j, i \rangle \in \mathit{vote\_msg}, \\
&\quad \langle i, k \rangle \in \mathit{vote\_msg} \\
&\quad i = j, i = k, j = k, \\
&\quad i \in \mathit{votes}[j], i \in \mathit{votes}[k], \\
&\quad j \in \mathit{votes}[i], j \in \mathit{votes}[k], \\
&\quad k \in \mathit{votes}[i], k \in \mathit{votes}[j], \\
&\quad \mathit{leader}[i], \\
&\quad \mathit{leader}[j], \\
&\quad \mathit{leader}[k], \\
&\quad \mathit{vi} \in \mathit{decided}[i], \mathit{vi} \in \mathit{decided}[j], \\
&\quad Q = \mathit{votes}[i], Q = \mathit{votes}[j] \qquad (\textsc{Preds}_1) \\
&\}
\end{aligned}
$$

(a) Full grammar for *SimpleConsensus*.

$$
\begin{aligned}
&\mathit{Pred}_{\{\mathit{leader},\mathit{decided}\}} \triangleq \{ \\
&\quad i = j, i = k, j = k \\
&\quad \mathit{leader}[i], \\
&\quad \mathit{leader}[j], \\
&\quad \mathit{leader}[k], \\
&\quad v \in \mathit{decided}[i], v \in \mathit{decided}[j], \\
&\quad \mathit{decided}[i] = \{\}, \mathit{decided}[j] = \{\} \\
&\}
\end{aligned}
$$

(b) Grammar slice for $\{leader, decided\}$.

$$
\begin{aligned}
&\mathit{Pred}_{\{\mathit{leader},\mathit{votes}\}} \triangleq \{ \\
&\quad i = j, i = k, j = k \\
&\quad \mathit{leader}[i], \\
&\quad \mathit{leader}[j], \\
&\quad \mathit{leader}[k], \\
&\quad i \in \mathit{votes}[j], i \in \mathit{votes}[k] \\
&\quad j \in \mathit{votes}[i], j \in \mathit{votes}[k] \\
&\quad k \in \mathit{votes}[i], k \in \mathit{votes}[j] \\
&\quad Q = \mathit{votes}[i], Q = \mathit{votes}[j] \\
&\}
\end{aligned}
$$

(c) Grammar slice for $\{votes, vote\_msg\}$.

Fig. 5. Full grammar and some grammar slices for *SimpleConsensus* protocol. Predicate sets in red indicate predicates removals that lead to inductive invariant inference failure for *SimpleConsensus*.

state set. The variable slices and sizes of the projected state sets at each node are shown in Figure 4. In our evaluation in Section 6, we demonstrate the effectiveness of these techniques on larger, more complex protocols, showing in several cases an order of magnitude improvement over existing syntax-guided approaches.

2.1.2 Interpretability and Failure Diagnosis. Notably, in addition to our technique taking advantage of the inductive proof graph structure for accelerating automated inference, the structure also provides a natural mechanism for fine-grained failure diagnosis and repair in cases where a complete inductive proof graph is not synthesized automatically. In particular, global inference failures manifest as failures localized to particular nodes of this graph e.g., when a local inference task cannot complete due to a timeout or some specified resource limit. Upon failure, a user can focus on inspecting this particular node of the graph, and its variable and grammar slice, to consider how the grammar may need to be repaired/modified in order for the overall inference procedure to succeed. For example, if the full *SimpleConsensus* grammar from Figure 5a is modified to remove the subset of predicates indicated as $\textsc{Preds}_1$, our inference algorithm may terminate having generated an incomplete, partial proof graph such as the one shown in Figure 3b. In this case, this failure is due to the fact that the grammar with $\textsc{Preds}_1$ removed is no longer is able to express properties about quorums garnered by leaders from voters, which is needed to synthesize a lemma like *LeaderHasQuorum* to support the *UniqueLeaders* lemma. By inspecting the action, lemma, and variable slice of the failed node, this provides a concrete and fine-grained mechanism for a user to interpret the automated failure and take effort to repair the grammar. This is a simple example of

the interpretability of our technique, but we examine this in greater detail, on larger protocols, in our evaluation in Section 6.

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

3 PRELIMINARIES

We are focused on the problem of safety verification of protocols formalized as discrete transition systems, which consists of a core problem of finding adequate inductive invariants. Furthermore, we are focused on verification of systems that are assumed to be correct i.e., we assume various bug-finding methods ([18],[3]) have been applied upfront before a proof is undertaken.

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

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

Inductive Invariants and Relative Induction. The standard technique for proving an invariant S of a system $M = (I, T)$ is to develop an *inductive invariant* [32], which is a state predicate Ind such that $Ind \Rightarrow S$ and

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

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

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

Typically, an inductive invariant is represented as a strengthening of S via a conjunction of smaller *lemma invariants*, $L_1, \dots, L_k$, such that the final inductive invariant is defined as $Ind = S \wedge L_1 \wedge \dots \wedge L_k$. Throughout this paper we assume inductive invariants can be represented in this form. Note also that for a given system $M = (I, T)$, a state predicate may be inductive only under the assumption

of some other predicate. For given state predicates Ind and L, if the formula $L \wedge Ind \wedge T \Rightarrow Ind'$ is valid, we say that Ind is *inductive relative to* L.

4 INDUCTIVE PROOF GRAPHS

Our inductive invariant inference technique is based around on 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 accelerating invariant inference via slicing, and also to localized reasoning and human interpretability.

4.1 Inductive Invariant Decomposition

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 [32]. 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

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

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

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

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

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

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 described in Section 3. 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 4.1. For a system $M = (I, T)$ with $T = A_1 \vee \cdots \vee A_k$, an *inductive proof graph* is a directed graph (V, E) where

- $V = V_L \cup V_A$ consists of a set of *lemma nodes* V_L and *action nodes* V_A, where
  - V_L is a set of state predicates over M.

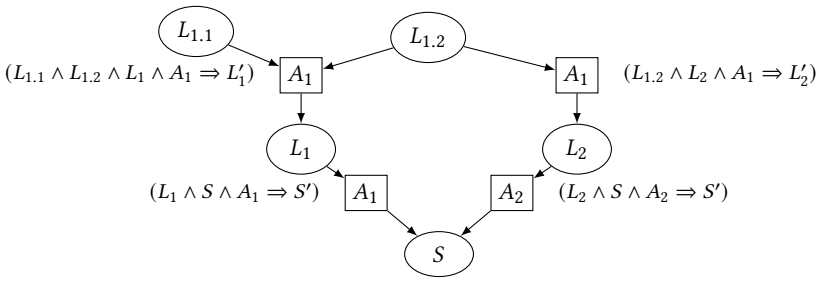


Fig. 6. Sample inductive proof graph. Lemma nodes are depicted as ovals, action nodes as boxes, and associated inductive proof obligations are shown in parentheses next to each action node. Self-inductive obligations are omitted for brevity. Action to lemma node relationships are shown as incoming lemma node edges.

  - $V_A = V_L \times \{A_1, \ldots, A_k\}$ is a set of action nodes, associated with each lemma node in V_L.
- $E \subseteq V_L \times V_A$ is a set of *lemma support edges*.

Figure 6 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.

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

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

Definition 4.3 (Local Lemma Validity). For an inductive proof graph $(V_L \cup V_A, E)$, a lemma node $L \in V_L$ is *locally valid* if all of its associated action nodes, $\{L\} \times \{A_1, \ldots, A_k\}$, are locally valid. 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 4.4 (Inductive Proof Graph Validity). An inductive proof graph is *valid* whenever all lemma nodes of the graph are *locally valid*.

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

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

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

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

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

4.2.1 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, consider a simple *ring counter* system with 3 state variables, a, b, and c, where a single value gets passed from a to b to c and exactly one variable holds the value at any time. An inductive invariant establishing the property that a always has a well-formed value will consist of 3 properties that form a 3-cycle, each stating that a,b and c's state are, respectively, always well-formed.

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

5 INVARIANT INFERENCE WITH INDUCTIVE PROOF SLICING

Our inductive invariant inference technique, *inductive proof slicing*, builds an invariant inference algorithm around the inductive proof graph as its core data structure. Our algorithm integrates the inductive proof graph with a syntax-guided inductive invariant synthesis approach similar to previously explored approaches [10, 39]. Explicit maintenance of the proof graph structure during inference, however, allows a series of localized *slicing* optimizations that accelerate local inference tasks by several orders of magnitude in many cases. Also, as discussed previously, it provides an effective, fine-grained interpretability and diagnosis mechanism when global inference fails to find a complete proof.

5.1 Our Invariant Inference Algorithm

At a high level, our inductive invariant inference algorithm incrementally constructs an inductive proof graph, starting from a given safety property S as its initial lemma node. It works backwards from the safety property by synthesizing support lemmas for any proof nodes that are not yet discharged. To synthesize these lemmas, we perform a local, syntax-guided invariant synthesis routine at each proof graph node. 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 finding an inductive invariant sufficient for proving S. It

Algorithm 1 Our inductive invariant inference algorithm with proof slicing.

```
1:  Inputs: Transition system M = (I, T), target safety property S, grammar Preds.
2:  procedure DoIndProofSlice(M, S, Preds)
3:      V_L ← {S}                                          ▷ Initialize the inductive proof graph.
4:      V_A ← {S} × {A_1, ..., A_k}
5:      E ← ∅
6:      G ← (V_L ∪ V_A, E)
7:      failed ← ∅
8:      if ∀a ∈ V_A : (a is locally valid) ∨ (a ∈ failed) then   ▷ Check if all graph nodes are discharged.
9:          return (G, failed).                            ▷ Returned graph G is valid if failed = ∅
10:     else
11:         Pick (L, A) ∈ (V_A \ failed) such that (L, A) is not locally valid.   ▷ Pick a graph node to work on.
12:         (Supp_(L,A), success) ← LocalInvInference(M, Preds, L, A)   ▷ See Algorithm 2
13:         if ¬success then
14:             failed ← failed ∪ {(L, A)}
15:             goto Line 8
16:         end if
17:         V_L ← V_L ∪ Supp                               ▷ Update the proof graph.
18:         V_A ← V_A ∪ (Supp × {A_1, ..., A_k})
19:         E ← E ∪ (Supp × {(L, A)})
20:         goto Line 8.
21:     end if
22: end procedure
```

starts by initializing an inductive proof graph $(V_L \cup V_A, E)$ where $V_L = \{S\}$, $V_A = \{S\} \times \{A_1, \ldots, A_k\}$, and $E = \emptyset$, as shown on Line 3 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, LocalInvInference, of the overall algorithm, and is shown separately as Algorithm 2, and described in more detail below in Section 5.2. Once the local inference call LocalInvInference 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 inference 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 make it efficient, 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.

Algorithm 2 Local invariant synthesis routine.

```
1: procedure LocalInvInference(M, Preds, L, A)
2:     Vars_(L,A) ← Slice(L, A)                          ▷ See Definition 5.2.
3:     Preds_(L,A) ← GrammarSlice(Preds, Vars_(L,A))     ▷ Filters grammar to predicates with variables in Vars_(L,A).
4:     Supp_(L,A) ← ∅
5:     CTIs ← GenerateCTIs(M, L, A)                      ▷ Generate local counterexamples to induction (violations of L ∧ A ⇒ L′).
6:     while CTIs ≠ ∅ do
7:         Invs ← GenerateLemmaInvs(M, Vars_(L,A), Preds_(L,A))
8:         if ∃A ∈ Invs : A eliminates some CTI in CTIs then
9:             Pick L_max ∈ Invs that eliminates the most CTIs from CTIs.
10:            Supp_(L,A) ← Supp_(L,A) ∪ {L_max}
11:            CTIs ← CTIs \ {s ∈ CTIs : s ⊭ L_max}
12:        else
13:            either goto Line 7
14:            or return (Supp_(L,A), False)              ▷ Fail: couldn't eliminate all CTIs.
15:        end if
16:    end while
17:    return (Supp_(L,A), True)                          ▷ Success: eliminated all CTIs
18: end procedure
```

5.2 Local Inference and Variable Slicing

As discussed above, our local inference routine, LocalInvInference, shown in Algorithm 2, is executed at each graph node (L, A). It begins by computing a local variable slice, $Vars_{(L,A)}$, at this node (Algorithm 2, Line 2), which is a subset of the overall state variables of the given system M that are sufficient to consider when discharging this node. Next, a local grammar slice (Line 3 of Algorithm 2) is computed based on this variable slice, and then we generate a set of local counterexamples to induction (CTIs), which are counterexamples to the local inductive proof obligation, $L \wedge A \Rightarrow L'$. The main local inference loop then begins at Line 7, which involves the generation of candidate lemma invariants to use for elimination of the set of generated CTIs. This invariant generation procedure, defined as GenerateLemmaInvs in Algorithm 2, also makes use of the variable slice $Vars_{(L,A)}$ to accelerate its computations, which we discuss in more detail below in Section 5.3.

The above provides a high level overview of our local inference routine. Key to our overall approach is the acceleration of this routine based on the variable slices computed at each node, so we discuss below the computation of these slices more precisely. We then discuss how these slices are used to accelerate local inference in more detail in Section 5.3.

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

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

predicate over a set of current state variables, denoted $Vars(Pre) \subseteq \mathcal{V}$, and $Post$ is a conjunction of update expressions of the form $x_i' = f_i(\mathcal{D}_i)$, where $x_i' \in \mathcal{V}'$ and $f_i(\mathcal{D}_i)$ is an expression over a subset of current state variables $\mathcal{D}_i \subseteq \mathcal{V}$.

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

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

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

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

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

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

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

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

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

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

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

□

5.3 Accelerating Local Inference with Slicing

As described above and shown in Algorithm 2, our local inference algorithm consists of a main loop that searches for candidate protocol invariants to eliminate a set of locally generated CTIs. The search space for these candidate invariants is defined by the grammar given as input to our overall algorithm, *Preds*, and the GenerateLemmaInvs routine uses a set of reachable system states R to validate these candidate invariants. Thus, the number of candidates generates by *Preds* and the size of R are the two largest factors impacting the performance of these local inference 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 our *grammar slicing* and *state slicing* optimizations.

5.3.1 Grammar Slicing. At the beginning of local inference, we use our local variable slice to prune the set of predicates in this global grammar based on the variables that appear in the local variable slice. To compute the local grammar slice at a node (L, A), we simply filter out any atomic predicates in *Preds* that do not refer to a set of variables that is a subset of $Vars_{(L,A)}$, our locally computed variable slice. This local grammar slice is passed as input to our invariant enumeration routine, GenerateLemmaInvs, which samples candidate predicates generated by this grammar.

Aa a concrete example, in our *SimpleConsensus* protocol example, when using the grammars as shown in Figure 5a, when operating on the $\{leader, decided\}$ variable slice, our technique generates a set of 3 term candidate invariants over the grammar slice in 5b of size 448, after applying basic logical equivalence reductions. Without grammar slicing applied (i.e., using the full grammar of 5a), we generate 18,312 candidates, a roughly 40x reduction in the search space of candidates enabled by grammar slicing in this case.

5.3.2 State Slicing. After generating candidate invariants defined by a local grammar slice, the GenerateLemmaInvs routine uses a set of reachable system states R to validate these candidate invariants. In general, this set of explored reachable states sampled during inference, R, only needs to be computed once e.g., at the beginning of Algorithm 1, since the underlying system M is fixed and does not change throughout inference. Furthermore, when we run a local inference routine, we both compute a local projection of this state set R, projecting out any variables absent from the local variable slice $Vars_{(L,A)}$, and cache this projected state set for future use throughout the algorithm.

For a counterexample guided inference procedure like ours, there are often several rounds of inference that occur, so this set R may be re-used many times, at different local inference tasks throughout. So, even with some upfront cost paid for caching this set and sets of sliced projections, computation of these projected state sets can often significantly speed up inference for large tasks, as we build up a cache of projected state sets. We show in our evaluation how this can often have a significant impact on the efficiency of the invariant inference procedure, since we often reduce the state space to check invariants over by several orders of magnitude.

For example, as discussed for the *SimpleConsensus* example protocol presented in Section 2, it contains 6 state variables, and for finite parameter instantiations where $|Node| = 3$ and $|Value| = 2$, the set of explored reachable states is 110,464. Since a variable slice is simply a subset of state variables, with 6 state variables there are $64 = 2^6$ possible slices. The 75th percentile of these state set sizes is 28,415, a $\approx$ 4x reduction from the full set of 110,464 states. In practice, as shown in Figure 4, the actual size of these projected state slices used during verification can often be significantly smaller as a fraction of the original set R.

5.4 Interpretability and Failure Diagnosis

Our inference algorithm constructs an inductive proof graph as it proceeds, and in cases of failure to generate a complete, valid proof graph, this structure provides a natural way to understand what parts of the inference procedure were problematic. In particular, failures are localized to specific nodes of this graph, which correspond to specific lemma and action pairs, along with the variable slice computed at that node, and the corresponding grammar slice. Although we don't have a fully automatic repair procedure, this localization provides guidance to a user if their aim is to repair the grammar to achieve convergence. Specifically, it allows them to focus on (1) the particular subset of variables in the node's slice and (2) the action and lemma of the failed node. In practice, this often provides useful guidance in terms of understanding why the tool wasn't able to synthesize a complete inductive proof.

Furthermore, a user can also attempt to examine counterexamples to induction (CTIs) at this local node that were not eliminated by existing set of synthesized support lemma $Supp_{(L,A)}$ as shown in Algorithm 1. We illustrate this in more depth on a concrete protocol example in our evaluation section, demonstrating how this can be an effective approach in aiding convergence for complex protocol verification tasks.

6 EVALUATION

We evaluated our technique along both the dimensions of *scalability* and *interpretability*. To evaluate scalability, 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. Some of the larger benchmarks we test include models of the Hermes and Zeus replication protocols [23, 24], which are modern, nontrivial fault-tolerant protocols that were published with accompanying formal specifications. We also test a large, asynchronous, message-passing specification of the Raft consensus algorithm [35], and models of the Bakery algorithm, a concurrent mutual exclusion algorithm [27]. We compare our approach against another state of the art inductive invariant inference technique that is based on a similar, syntax-guided synthesis approach and accepts similar inputs [39].

To examine the interpretability of our technique, we present a case study of examining some of the proof graphs synthesized by our tool, and how these graphs aided our understanding and development of synthesizing an inductive invariant for our message passing Raft benchmark.

Implementation and Setup. Our inductive invariant inference algorithm is implemented in a tool, Scimitar, which consists of approximately 6100 lines of Python code, and accepts as input protocols specified in the TLA+ specification language [29]. 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 [30]. 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. Code for our implementation and protocol benchmarks and results described below is available in the supplementary material for this paper, along with instructions for running our benchmarks.

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. We also use fixed parameters to our tool, which include a setting of $N_{invs} = 80,000$, which defines the number of candidate invariants sampled at each inference round, and $N_{ctis} = 10,000$, which defines the maximum number of CTIs to generate at each round to use for elimination checking.

6.1 Scalability

6.1.1 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 medium-large protocols, and to also examine the scale of protocols it could solve that existing tools could not.

						Scimitar		endive
Benchmark	LoC	$\|R\|$	$Vars$	$\|A\|$	$\|Preds\|$	# Lemmas	Time	Time
TwoPhase	195	288	6	7	18	12	348	173
SimpleConsensus	108	110,464	6	5	25	9	748	416
ZeusReliableCommit	363	339,985	11	11	70	7	997	4903
Hermes	355	2,500,000	11	9	90	6	477	timeout
Bakery	234	6,016,610	6	10	60	19	6121	timeout
AsyncRaft	583	10,000,000	12	15	110	9	13524	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 infer an inductive invariant. The (endive) column represents the baseline approach based on the technique of [39], when run with the same relevant parameters. A *timeout* entry indicates no invariant found after an 8 hour timeout.

The *TwoPhase* benchmark is a high level specification of the two-phase commit protocol [15], and *SimpleConsensus* is the consensus protocol presented in Section 2. The *Hermes* and *ZeusReliableCommit* benchmarks are specifications of modern replication protocols whose designs draw inspiration from cache coherence protocols [23, 24]. Both protocols were published recently and were originally presented with TLA+ specifications, which we use in our benchmarks.

The *Bakery* benchmark is a specification of Lamport's Bakery algorithm for mutual exclusion [27]. The largest benchmark tested is an industrial scale specification of the Raft consensus protocol [35]. The specification we use is based on a model similar to the original Raft formal specification [34, 44], and models asynchronous message passing between all nodes and fine-grained local state. The safety property checked is the high level *ElectionSafety* property of Raft, which states that no two leaders can be elected in the same "term" value, which is a monotonic integral timestamp maintained at each server in the Raft protocol.

We note that the largest protocol specifications we test are 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 scalability 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 [37], 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 inference 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].

6.1.2 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 [39], 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

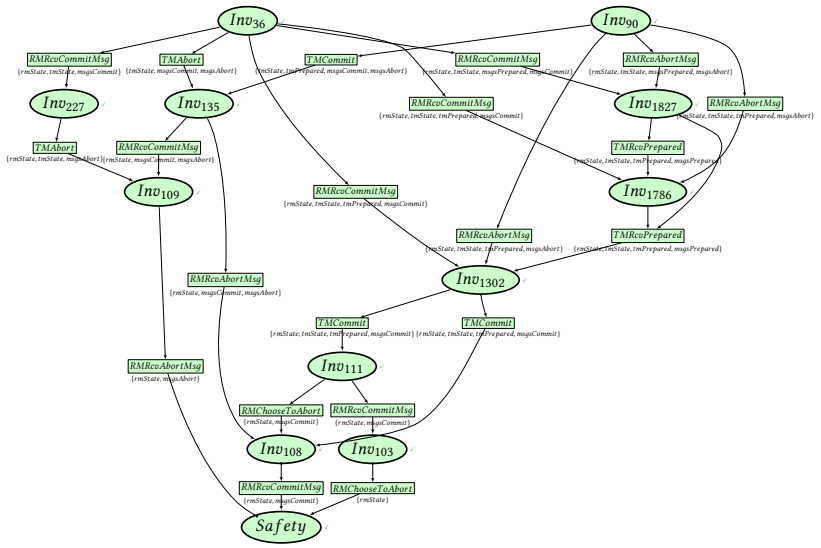


Fig. 7. Complete, synthesized inductive proof graph for the *TwoPhase* specification. the safety property labeled as *Safety* is the consistency property of two-phase commit stating that no two resource managers have conflicting commit and abort decisions. Local variable slices are shown as sets below each action node.

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 [39]. 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 states initially, etc. Also, as an artifact of our current implementation, each local inference on the proof graph can have a slightly higher startup overhead than the *endive* implementation, which batches and parallelizes more inference tasks together. 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.

At the medium protocol level, the baseline approach of *endive* can still solve some benchmarks, but our technique provides an order of magnitude performance improvement. For example, for the *ZeusReliableCommit* benchmark, we see a roughly 5x improvement in runtime. For *Hermes*, the *endive* tool was not able to find an inductive invariant after an 8 hour timeout, and ours was able to find one in less than 10 minutes. Note that the size of the *Hermes* protocol's reachable state space is too large to instantiate with reasonable finite parameters that can be fully exhausted, so the $|R|$ column represents a set of 2.5 million reachable states that we sampled for this inference run. The performance of the Hermes benchmark in this case correlates strongly with the effectiveness of state slicing in this case. For example, though the set R is over 2 million states for this benchmark, during our inference run, the largest state slice used for local inference was 36,418 states ($\approx$ 40x reduction), based on a largest variable slice containing 4 / 11 total state variables.

In the largest class of protocols, including *Bakery* and *AsyncRaft*, our approach is able to solve all of these benchmarks whereas *endive* times out on all of them. The *AsyncRaft* protocol is the largest

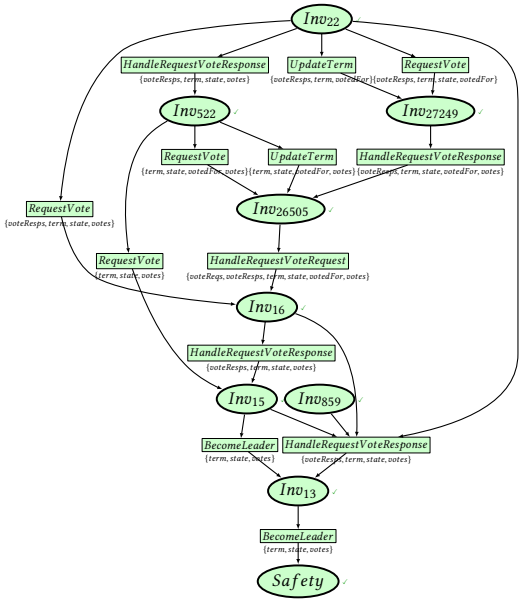


Fig. 8. Synthesized *AsyncRaft* inductive proof graph for proof of the *ElectionSafety* property.

benchmark we test, and we are able to synthesize an inductive proof graph for the *ElectionSafety* property in just under 4 hours. We sample a set of 10,000,000 reachable states for the set $|R|$ for this benchmark, and, similarly, the performance is heavily impacted by the gains from state slicing during these inference runs. We observed that the largest state slice set used during a local inference run for this benchmark was approximately 18,183 states, again an order of magnitude reduction over the full set of states from R. Our approach is also seen to be effective at intelligently focusing on subsets of variables that are relevant for each local inference task. For example, Figure 8 shows the complete inductive proof graph synthesized for *AsyncRaft* and the *ElectionSafety* property, and its variable slices are often relatively small subsets of the overall state variable set of 12 variables for the *AsyncRaft* protocol specification.

6.2 Interpretability

To examine the interpretability of our technique, we analyzed how our inductive proof graph structure aided in the development and understanding of inductive proof graphs, and the repair of grammars in cases of failure on real protocols.

As a high level presentation of the interpretability of our method, Figure 7 shows a complete synthesized proof graph for the *TwoPhase* benchmark, which is a specification of the classic two-phase commit protocol [15]. In the two-phase commit protocol, a transaction manager aims to achieve agreement from a set of resource managers on whether to *commit* or *abort* a transaction. This proof graph establishes the core safety property of two-phase commit which states that no two resource managers can come to conflicting commit and abort decisions. The structure of this proof graph provides an intuitive way to understand the structure of the inductive proof in a way that a monolithic inductive invariant does not. For example, it admits a relatively tree-like structure, and we can naturally focus on sub-components on the graph, and consider parts of the graph based on actions and their relationships to lemmas. For example, it is intuitive to see that the root safety node, *Safety*, has support lemmas via 2 actions, *RMRcvCommitMsg* and *RMChooseToAbort*, which

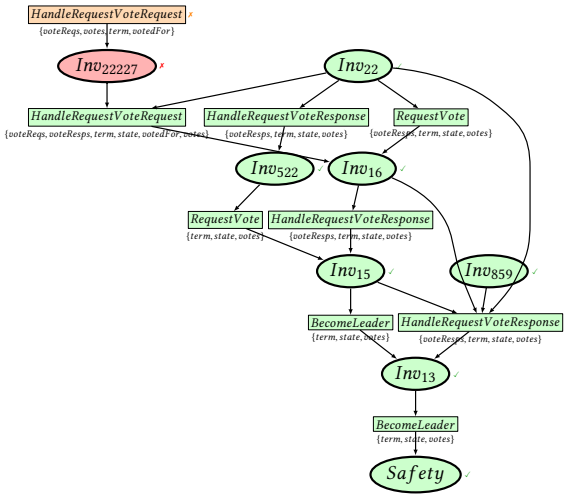


Fig. 9. Incomplete *AsyncRaft* inductive proof graph for proof of the *ElectionSafety* property that was synthesized when missing grammar predicates relating to vote request messages (*voteReqs*) and the locally recorded votes on servers (*votedFor*).

represent, respectively, the actions a resource manager takes to commit or abort. That is, these are specifically the actions that can directly falsify the target safety property of two-phase commit. More precisely, lemma Inv_{108} states the invariant that if a resource manager has aborted, then there cannot have been a "commit" message sent by the transaction manager, which is necessary to ensure that the safety is not violated by some resource manager trying to commit.

We additionally examined how the interpretability of inductive proof graphs can be used in assisting our tool when fully automated synthesis initially fails. We studied this in the context of the development of the *AsyncRaft* benchmark. In Figure 8, we show a complete synthesized proof graph for the high level *ElectionSafety* property of Raft, which states that two leaders cannot be elected in the same "term", which is a monotonic counter maintained at each server in the Raft protocol.

During development of this proof, we encountered a case of partial convergence, where an incomplete proof graph was synthesized with some failed nodes. For example, Figure 9 shows an example of such a partial proof graph for the *AsyncRaft* protocol, where the proof graph failed to synthesize an inductive proof for the target *ElectionSafety* property. In this case, there was a synthesized lemma, Inv_{22227}, whose failure ultimately was caused by the absence of predicates in our grammar that related the *votedFor* variable of Raft, which locally records whether a server has voted for another server, and the set of vote request messages in flight, *voteReqs*, both of which appear in the local variable slice of the failed node, Inv_{22227}. Small modifications to the grammar based on these observations were part of the development process that led us to converging on a complete proof graph as shown in Figure 8.

Overall, we found that these proof graph structures generally provided a much greater degree of insight into and understanding of the synthesis process versus other, monolithic approaches. In particular, they were helpful in understanding the intuitive structure of a protocol and aiding in understanding what sections of the protocol our automated tool was finding difficult to handle.

7 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 [12], SWISS [17] DistAI [48], and others [39, 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 existing modern tools in this area can solve.

In the related domain of general model checking algorithms, we believe that our inductive proof graph structure for managing large scale inductive invariants bears similarities with approaches developed for managing proof obligation queues in IC3/PDR [2, 16]. In some sense, our approach revolves around making the set of proof obligations and their dependencies explicit (and also incorporating action-based decomposition), which can be a key factor in tuning of IC3/PDR, which often has many non-deterministic choices throughout execution.

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 [40].

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

In the Ivy system [37] 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 [22]. 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 [1, 33].

Concurrent Program Analysis. Our techniques presented in this paper bear similarities to prior approaches used in the analysis and proofs of concurrent programs. Our notion of inductive proof graphs is similar to the *inductive data flow graph* concept presented in [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 [14], as well as other *program slicing* techniques [43]. It also shares some concepts with other path-based program analysis techniques that incorporate slicing techniques [20, 21]. In our case, however, we apply it at the level of a single protocol action and target lemma, particularly for the purpose of accelerate syntax-guided invariant synthesis tasks.

8 CONCLUSIONS AND FUTURE WORK

We presented *inductive proof slicing*, a new automated technique for inductive invariant inference of large scale distributed protocols that scales to protocols more complex than those handled by existing techniques. 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.

DATA-AVAILABILITY STATEMENT

Code for our implementation and protocol benchmarks and results described in Section 6 is available in the supplementary material for this paper.

REFERENCES

[1] 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.

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

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

[4] 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.

[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., 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.

[13] 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.

[14] 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.

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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).

[23] Katsarakis, A., Gavrielatos, V., Katebzadeh, M. S., Joshi, A., Dragojevic, A., Grot, B., and Nagarajan, V. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 201–217.

[24] Katsarakis, A., Ma, Y., Tan, Z., Bainbridge, A., Balkwill, M., Dragojevic, A., Grot, B., Radunovic, B., and Zhang, Y. Zeus: Locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems* (New York, NY, USA, 2021), EuroSys '21, Association for Computing Machinery.

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

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

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

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

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

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

[31] 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).

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

[33] McMillan, K. A methodology for hardware verification using compositional model checking. *Science of Computer*

Programming 37, 1 (2000), 279–309.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[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.