On Symmetry and Quantification: A New Approach to Verify Distributed Protocols

Aman Goel(✉) and Karem Sakallah

University of Michigan, Ann Arbor MI 48105, USA
{amangoel,karem}@umich.edu

Abstract. Proving that an unbounded distributed protocol satisfies a given safety property amounts to finding a quantified inductive invariant that implies the property for all possible instance sizes of the protocol. Existing methods for solving this problem can be described as search procedures for an invariant whose quantification prefix fits a particular template. We propose an alternative *constructive* approach that does not prescribe, *a priori*, a specific quantifier prefix. Instead, the required prefix is automatically inferred without any search by carefully analyzing the structural symmetries of the protocol. The key insight underlying this approach is that symmetry and quantification are closely related concepts that express protocol *invariance* under different re-arrangements of its components. We propose *symmetric incremental induction*, an extension of the finite-domain IC3/PDR algorithm, that automatically derives the required *quantified inductive invariant* by exploiting the connection between symmetry and quantification. While various attempts have been made to exploit symmetry in verification applications, to our knowledge, this is the first demonstration of a direct link between symmetry and quantification in the context of clause learning during incremental induction. We also describe a procedure to automatically find a minimal finite size, the *cutoff*, that yields a quantified invariant proving safety for any size.

Our approach is implemented in IC3PO, a new verifier for distributed protocols that significantly outperforms the state-of-the-art, scales orders of magnitude faster, and robustly derives compact inductive invariants fully automatically.

1 Introduction

Our focus in this paper is on *parameterized verification*, specifically proving *safety* properties of distributed systems, such as protocols that are often modeled above the code level (e.g., [49, 63]), consisting of arbitrary numbers of *identical* components that are instances of a small set of different *sorts*. For example, a client server protocol [1] $CS(i, j)$ is a two-sort parameterized system with parameters $i \geq 1$ and $j \geq 1$ denoting, respectively, the number of clients and servers. Protocol correctness proofs are critical for establishing the correctness of actual system implementations in established methodologies such as [42, 69]. Proving safety properties for such systems requires the derivation of inductive invariants that are expressed as state predicates quantified over the system parameters. While, in general, this problem is undecidable [8], certain restricted forms have been shown to yield to algorithmic solutions [17]. Key to these solutions is appealing to the problem's inherent symmetry. In this paper, we exclusively focus on protocols whose sorts represent sets of indistinguishable domain constants. The behavior of this restricted class of protocols remains invariant under all possible permutations of the domain constants. We leave the exploration of other features, such as totally-ordered sorts, integer arithmetic, etc., for future work.

Our proposed symmetry-based solution is best understood by briefly reviewing earlier efforts. Initially, the pressing issue was the inevitable *state explosion* when verifying a finite, but large, parameterized system [12, 29, 37, 60, 66, 68]. Thus, instead of verifying the "full" system, these approaches verified its *symmetry-reduced quotient,* mostly using BDD-based symbolic image computation [19,20,56]. The Murφ verifier [60] was a notable exception in that it a) generated a C++ program that enumerated the system's symmetry-reduced reachable states, and b) allowed for the verification of unbounded systems by taking advantage of *data saturation* which happens when the size of the symmetry-reduced reachable states become constant regardless of system size.

The idea that an unbounded *symmetric* system can, under certain data-independence assumptions, be verified by analyzing small finite instances evolved into the approach of verification by *invisible invariants* [9,10,25,65,70]. In this approach, assuming they exist, inductive invariants that are universally-quantified over the system parameters are automatically derived by analyzing instances of the system up to a *cutoff* size N_0 using a combination of symbolic reachability and symmetry-based abstraction. Noting that an invariant is an over-approximation of the reachable states, the restriction to universal quantification may fail in some cases, rendering the approach incomplete. The invisible invariant verifier IIV [10] employs some heuristics to derive invariants that use combinations of universal and existential quantifiers, but as pointed out in [58], it may still fail and is not guaranteed to be complete.

The development of SAT-based incremental induction algorithms [18, 27] for verifying the safety of finite transition systems was a major advance in the field of model checking and has, for the most part, replaced BDD-based approaches. These algorithms leverage the capacity and performance of modern CDCL SAT solvers [11, 28, 55, 57] to produce *clausal strengthening assertions* A that, conjoined with a specified safety property P, form an automatically-generated inductive invariant $Inv = A \wedge P$ if the property holds. The AVR hardware verifier [38–40] was adapted in [53] to produce quantifier-free inductive invariants for small instances of unbounded protocols that are subsequently generalized with universal quantification, in analogy with the invisible invariants approach, to arbitrary sizes. The resulting assertions tended, in some cases, to be quite large, and the approach was also incomplete due to the restriction to universal quantification.

In this paper we introduce IC3PO, a novel symmetry-based verifier that builds on these previous efforts while removing most of their limitations. Rather than search for an invariant with a prescribed quantifier prefix, IC3PO constructively *discovers* the required quantified assertions by performing *symmetric incremental induction* and analyzing the symmetry patterns in learned clauses to infer the corresponding quantifier prefix. Our main contributions are:

- An extension to finite incremental induction algorithms that uses protocol symmetry to boost clause learning from a *single* clause φ to a set of symmetrically-equivalent clauses, φ's *orbit*.
- A quantifier inference procedure that expresses φ's orbit by an automatically-derived *compact* quantified predicate Φ. The inference procedure is based on a simple analysis of φ's *syntactic structure* and yields a quantified form with both universal and existential quantifiers.
- A systematic *finite convergence* procedure for determining a minimal instance size sufficient for deriving a quantified inductive invariant that holds for all sizes.

We also demonstrate the effectiveness of IC3PO on a diverse set of benchmarks and show that it significantly advances the current state-of-the-art.

The paper is structured as follows: §2 presents preliminaries. §3 formalizes protocol symmetries. The next three sections detail our key contributions: symmetry boosting during incremental induction in §4, relating symmetry to quantification in §5, and checking for convergence in §6. §7 describes the IC3PO algorithm and implementation details. §8 presents our experimental evaluation. The paper concludes with a brief survey of related work in §9, and a discussion of future directions in §10.

2 Preliminaries

Figure 1 describes a toy consensus protocol from [6] in the TLA+ language [49].[1] The protocol has three named sorts $S = [\mathtt{node}, \mathtt{quorum}, \mathtt{value}]$ introduced by the CONSTANTS declaration, and two relations $R = \{vote, decision\}$, introduced by the VARIABLES declaration, that are defined on these sorts. Each of the sorts is understood to represent an unbounded domain of distinct elements with the relations serving as the protocol's state variables. The global axiom (line 3) defines the elements of the `quorum` sort to be subsets of the `node` sort and restricts them further by requiring them to be pair-wise non-disjoint. We will refer to `node` (resp. `quorum`) as an *independent* (resp. *dependent*) sort. The protocol transitions are specified by the actions *CastVote* and *Decide* (lines 6-7) which are expressed using the current- and next-state variables as well as the definitions *didNotVote* and *chosenAt* (lines 4-5) which serve as *auxiliary non-state* variables. Lines 8-10 specify the protocol's initial states, transition relation, and safety property.

Viewed as a parameterized system, the *template* of an arbitrary $\mathtt{n}$-sort distributed protocol $\mathcal{P}$ will be expressed as $\mathcal{P}(\mathtt{s}_1, \ldots, \mathtt{s}_n)$ where $S = [\mathtt{s}_1, \ldots, \mathtt{s}_n]$ is an ordered list of its sorts, each of which is assumed to be an unbounded uninterpreted set of distinct *constants*. As a mathematical transition system, $\mathcal{P}$ is defined by a) its state variables which are expressed as k-ary relations on its

[1] The description in [6] is in the Ivy [63] language and encodes set operations in relational form with a *member* relation representing $\in$.

MODULE *ToyConsensus*

CONSTANTS $\texttt{node}, \texttt{quorum}, \texttt{value}$ VARIABLES $vote, decision$

$vote \in (\texttt{node} \times \texttt{value}) \rightarrow \text{BOOLEAN}$ $decision \in \texttt{value} \rightarrow \text{BOOLEAN}$

ASSUME $\forall Q \in \texttt{quorum} : Q \subseteq \texttt{node} \;\wedge\; \forall Q_1, Q_2 \in \texttt{quorum} : Q_1 \cap Q_2 \neq \{\}$

$didNotVote(n) \triangleq \forall V \in \texttt{value} : \neg vote(n, V)$

$chosenAt(q, v) \triangleq \forall N \in q : vote(N, v)$

$CastVote(n, v) \triangleq didNotVote(n) \;\wedge\; vote' = [vote \text{ EXCEPT } ![n, v] = \text{TRUE}]$
$\qquad \wedge\; \text{UNCHANGED } decision$

$Decide(q, v) \triangleq chosenAt(q, v) \;\wedge\; decision' = [decision \text{ EXCEPT } ![v] = \text{TRUE}]$
$\qquad \wedge\; \text{UNCHANGED } vote$

$Init \triangleq \forall N \in \texttt{node}, V \in \texttt{value} : \neg vote(N, V) \;\wedge\; \forall V \in \texttt{value} : \neg decision(V)$

$T \triangleq \exists N \in \texttt{node}, Q \in \texttt{quorum}, V \in \texttt{value} : CastVote(N, V) \vee Decide(Q, V)$

$P \triangleq \forall V_1, V_2 \in \texttt{value} : decision(V_1) \wedge decision(V_2) \Rightarrow V_1 = V_2$

Fig. 1: Toy consensus protocol in the TLA+ language

sorts, and b) its actions which capture its state transitions. We also note that non-Boolean functions/variables can be easily accommodated by encoding them in relational form, e.g., $f(\texttt{x}_1, \texttt{x}_2, \dots) = \texttt{y}$. We will use *Init*, T, and P to denote, respectively, a protocol's initial states, its transition relation, and a safety property that is required to hold on all reachable states. A finite instance of $\mathcal{P}$ will be denoted as $\mathcal{P}(|\texttt{s}_1|, \dots, |\texttt{s}_\texttt{n}|)$ where each named sort is replaced by its finite size in the instance. Similarly, $Init(|\texttt{s}_1|, \dots, |\texttt{s}_\texttt{n}|)$, $T(|\texttt{s}_1|, \dots, |\texttt{s}_\texttt{n}|)$ and $P(|\texttt{s}_1|, \dots, |\texttt{s}_\texttt{n}|)$ will, respectively, denote the application of *Init*, T and P to this finite instance.

The template of the protocol in Figure 1 is $ToyConsensus(\texttt{node}, \texttt{quorum}, \texttt{value})$. Its finite instance:

$$ToyConsensus(3,3,3): \quad \texttt{node}_3 \triangleq \{\texttt{n}_1, \texttt{n}_2, \texttt{n}_3\} \qquad \texttt{value}_3 \triangleq \{\texttt{v}_1, \texttt{v}_2, \texttt{v}_3\} \tag{1}$$
$$\texttt{quorum}_3 \triangleq \{\texttt{q}_{12} : \{\texttt{n}_1, \texttt{n}_2\},\ \texttt{q}_{13} : \{\texttt{n}_1, \texttt{n}_3\},\ \texttt{q}_{23} : \{\texttt{n}_2, \texttt{n}_3\}\}$$

will be used as a running example in the paper. The finite sorts of this instance are defined as sets of arbitrarily-named distinct constants. It should be noted that the constants of the $\texttt{quorum}_3$ sort are subsets of the $\texttt{node}_3$ sort that satisfy the non-empty intersection axiom and are named to reflect their symmetric dependence on the $\texttt{node}_3$ sort. This instance has 9 *vote* and 3 *decision* state variables, and a *state* of this instance corresponds to a complete Boolean assignment to these 12 state variables.

In the sequel, we will use $\hat{\mathcal{P}}$ and $\hat{T}$ as shorthand for $\mathcal{P}(|\texttt{s}_1|, \dots, |\texttt{s}_\texttt{n}|)$ and $T(|\texttt{s}_1|, \dots, |\texttt{s}_\texttt{n}|)$. Quantifier-free formulas will be denoted by lower-case Greek letters (e.g., φ) and quantified formulas by upper-case Greek letters (e.g., Φ). We use primes (e.g., φ') to represent a formula after a single transition step.

3 Protocol Symmetries

The symmetry group of $\hat{\mathcal{P}}$ is $G(\hat{\mathcal{P}}) = \bigtimes_{\texttt{s} \in S} Sym(\texttt{s})$, where $Sym(\texttt{s})$ is the symmetric group, i.e., the set of $|\texttt{s}|!$ permutations of the constants of the set

$\mathtt{s}$.[2] In what follows we will use G instead of $G(\hat{\mathcal{P}})$ to reduce clutter. Given a permutation $\gamma \in G$ and an arbitrary protocol relation ρ instantiated with specific sort constants, the *action* of γ on ρ, denoted ρ^{γ}, is the relation obtained from ρ by permuting the sort constants in ρ according to γ; it is referred to as the γ-*image* of ρ. Permutation $\gamma \in G$ can also act on any formula involving the protocol relations. In particular, the invariance of protocol behavior under permutation of sort constants implies that the action of γ on the (finite) initial state, transition relation, and property formulas causes a syntactic re-arrangement of their subformulas while preserving their logical equivalence:

$$\hat{Init}^{\gamma} \equiv \hat{Init} \qquad \hat{T}^{\gamma} \equiv \hat{T} \qquad \hat{P}^{\gamma} \equiv \hat{P} \tag{2}$$

Consider next a clause φ which is a disjunction of literals, namely, instantiated protocol relations or their negations. The *orbit* of φ under G, denoted φ^{G}, is the set of its images φ^{γ} for all permutations $\gamma \in G$, i.e., $\varphi^{G} = \{\varphi^{\gamma} | \gamma \in G\}$. The γ-image of a clause can be viewed as a *syntactic* transformation that will either yield a new logically-distinct clause on different literals or simply re-arrange the literals in the clause without changing its logical behavior (by the commutativity and associativity of disjunction). We define the *logical action* of a permutation γ on a clause φ, denoted $\varphi^{L(\gamma)}$, as:

$$\varphi^{L(\gamma)} = \begin{cases} \varphi^{\gamma} & \text{if } \varphi^{\gamma} \not\equiv \varphi \\ \varphi & \text{if } \varphi^{\gamma} \equiv \varphi \end{cases}$$

and the *logical orbit* of φ as $\varphi^{L(G)} = \left\{ \varphi^{L(\gamma)} \middle| \gamma \in G \right\}$. With a slight abuse of notation, logical orbit can also be viewed as the conjunction of the logical images:

$$\varphi^{L(G)} = \bigwedge_{\gamma \in G} \varphi^{L(\gamma)}$$

To illustrate these concepts, consider *ToyConsensus*$(3, 3, 3)$ from (1). Its symmetries in cycle notation are as follows:

$$\begin{aligned} \mathit{Sym}(\mathtt{node}_3) &= \{(), (\mathtt{n}_1\ \mathtt{n}_2), (\mathtt{n}_1\ \mathtt{n}_3), (\mathtt{n}_2\ \mathtt{n}_3), (\mathtt{n}_1\ \mathtt{n}_2\ \mathtt{n}_3), (\mathtt{n}_1\ \mathtt{n}_3\ \mathtt{n}_2)\} \\ \mathit{Sym}(\mathtt{value}_3) &= \{(), (\mathtt{v}_1\ \mathtt{v}_2), (\mathtt{v}_1\ \mathtt{v}_3), (\mathtt{v}_2\ \mathtt{v}_3), (\mathtt{v}_1\ \mathtt{v}_2\ \mathtt{v}_3), (\mathtt{v}_1\ \mathtt{v}_3\ \mathtt{v}_2)\} \\ G = {} & \mathit{Sym}(\mathtt{node}_3) \times \mathit{Sym}(\mathtt{value}_3) \end{aligned} \tag{3}$$

The symmetry group (3) of *ToyConsensus*$(3, 3, 3)$ has 36 symmetries corresponding to the 6 $\mathtt{node}_3$ × 6 $\mathtt{value}_3$ permutations. The permutations on $\mathtt{quorum}_3$ are *implicit* and based on the permutations of $\mathtt{node}_3$ since $\mathtt{quorum}_3$ is a dependent sort. Now, consider the example clause:

$$\varphi_1 = \mathit{vote}(\mathtt{n}_1, \mathtt{v}_1) \lor \mathit{vote}(\mathtt{n}_1, \mathtt{v}_2) \lor \mathit{vote}(\mathtt{n}_1, \mathtt{v}_3) \tag{4}$$

The orbit of φ_1 consists of 36 syntactically-permuted clauses. However, many of these images are logically equivalent yielding the following logical orbit of just 3 logically-distinct clauses:

$$\begin{aligned} \varphi_1^{L(G)} = {} & [\ \mathit{vote}(\mathtt{n}_1, \mathtt{v}_1) \lor \mathit{vote}(\mathtt{n}_1, \mathtt{v}_2) \lor \mathit{vote}(\mathtt{n}_1, \mathtt{v}_3)\ ] \land \\ & [\ \mathit{vote}(\mathtt{n}_2, \mathtt{v}_1) \lor \mathit{vote}(\mathtt{n}_2, \mathtt{v}_2) \lor \mathit{vote}(\mathtt{n}_2, \mathtt{v}_3)\ ] \land \\ & [\ \mathit{vote}(\mathtt{n}_3, \mathtt{v}_1) \lor \mathit{vote}(\mathtt{n}_3, \mathtt{v}_2) \lor \mathit{vote}(\mathtt{n}_3, \mathtt{v}_3)\ ] \end{aligned} \tag{5}$$

[2] We assume familiarity with basic notions from *group theory* including *permutation groups*, *cycle notation*, *group action* on a set, *orbits*, etc., which can be readily found in standard textbooks on Abstract Algebra [33].

4 *SymIC3*: Symmetric Incremental Induction

SymIC3 is an extension of the standard IC3 algorithm [18,27] that takes advantage of the symmetries in a finite instance $\hat{\mathcal{P}}$ of an unbounded protocol $\mathcal{P}$ to *boost learning* during backward reachability. Specifically, it refines the current frame, in a *single* step, with *all* clauses in the logical orbit $\varphi^{L(G)}$ of a newly-learned quantifier-free clause φ. In other words, having determined that the backward 1-step check $F_{i-1} \wedge \hat{T} \wedge [\neg\varphi]'$ is unsatisfiable (i.e., that states in cube $\neg\varphi$ in frame F_i are unreachable from the previous frame F_{i-1}), *SymIC3* refines F_i with $\varphi^{L(G)}$, i.e., $F_i := F_i \wedge \varphi^{L(G)}$, rather than with just φ. Thus, at each refinement step, *SymIC3* not only blocks cube $\neg\varphi$, but also all symmetrically-equivalent cubes $[\neg\varphi]^\gamma$ for all $\gamma \in G$. This simple change to the standard incremental induction algorithm significantly improves performance since the extra clauses used to refine F_i a) are derived *without* making additional backward 1-step queries, and b) provide stronger refinement in each step of backward reachability leading to faster convergence with fewer counterexamples-to-induction (CTIs). The proof of correctness of symmetry boosting can be found in Appendix B.1.

5 Quantifier Inference

The key insight underlying our overall approach is that the explicit logical orbit, in a finite protocol instance, of a learned clause φ can be exactly, and systematically, captured by a corresponding quantified predicate Φ. In retrospect, this should not be surprising since symmetry and quantification can be seen as different ways of expressing invariance under permutation of the sort constants in the clause. To motivate the connection between symmetry and quantification, consider the following quantifier-free clause from our running example and a proposed quantified predicate that *implicitly* represents its logical orbit:

$$\begin{aligned}
&\varphi_2 = \neg \mathit{decision}(\mathtt{v}_1) \vee \mathit{decision}(\mathtt{v}_2)\\
&\Phi_2 = \forall X_1, X_2 \in \mathtt{value}_3 : \ (\text{distinct } X_1\ X_2) \rightarrow [\ \neg \mathit{decision}(X_1) \vee \mathit{decision}(X_2)\ ]
\end{aligned} \quad (6)$$

As shown in Table 1, the logical orbit $\varphi_2^{L(G)}$ consists of 6 logically-distinct clauses corresponding to the 6 permutations of the 3 constants of the $\mathtt{value}_3$ sort. Evaluating Φ_2 by substituting all $3 \times 3 = 9$ assignments to the variable pair $(X_1, X_2) \in \mathtt{value}_3 \times \mathtt{value}_3$ yields 9 clauses, 3 of which (shown faded) are trivially true since their "distinct" antecedents are false, with the remaining 6 corresponding to each of the clauses obtained through permutations of the 3 $\mathtt{value}_3$ constants. Similarly, we can show that the 3-clause logical orbit $\varphi_1^{L(G)}$ in (5) can be succinctly expressed by the quantified predicate:

$$\Phi_1 = \ \forall Y \in \mathtt{node}_3,\ \exists X \in \mathtt{value}_3 : \ \mathit{vote}(Y, X) \quad (7)$$

which employs universal *and* existential quantification. And, finally, φ_3 and Φ_3 below illustrate how a clause whose logical orbit is just itself can also be expressed as an existentially-quantified predicate.

$$\begin{aligned}
&\varphi_3 = \mathit{decision}(\mathtt{v}_1) \vee \mathit{decision}(\mathtt{v}_2) \vee \mathit{decision}(\mathtt{v}_3)\\
&\Phi_3 = \exists\ X \in \mathtt{value}_3 : \ \mathit{decision}(X)
\end{aligned} \quad (8)$$

(X_1, X_2)	Instantiation of Φ_2	Permutation
$(\mathtt{v}_1, \mathtt{v}_1)$	(distinct $\mathtt{v}_1$ $\mathtt{v}_1$) $\rightarrow$ [$\neg decision(\mathtt{v}_1) \vee decision(\mathtt{v}_1)$]	none
$(\mathtt{v}_1, \mathtt{v}_2)$	(distinct $\mathtt{v}_1$ $\mathtt{v}_2$) $\rightarrow$ [$\neg decision(\mathtt{v}_1) \vee decision(\mathtt{v}_2)$]	()
$(\mathtt{v}_1, \mathtt{v}_3)$	(distinct $\mathtt{v}_1$ $\mathtt{v}_3$) $\rightarrow$ [$\neg decision(\mathtt{v}_1) \vee decision(\mathtt{v}_3)$]	$(\mathtt{v}_2\ \mathtt{v}_3)$
$(\mathtt{v}_2, \mathtt{v}_1)$	(distinct $\mathtt{v}_2$ $\mathtt{v}_1$) $\rightarrow$ [$\neg decision(\mathtt{v}_2) \vee decision(\mathtt{v}_1)$]	$(\mathtt{v}_1\ \mathtt{v}_2)$
$(\mathtt{v}_2, \mathtt{v}_2)$	(distinct $\mathtt{v}_2$ $\mathtt{v}_2$) $\rightarrow$ [$\neg decision(\mathtt{v}_2) \vee decision(\mathtt{v}_2)$]	none
$(\mathtt{v}_2, \mathtt{v}_3)$	(distinct $\mathtt{v}_2$ $\mathtt{v}_3$) $\rightarrow$ [$\neg decision(\mathtt{v}_2) \vee decision(\mathtt{v}_3)$]	$(\mathtt{v}_1\ \mathtt{v}_2\ \mathtt{v}_3)$
$(\mathtt{v}_3, \mathtt{v}_1)$	(distinct $\mathtt{v}_3$ $\mathtt{v}_1$) $\rightarrow$ [$\neg decision(\mathtt{v}_3) \vee decision(\mathtt{v}_1)$]	$(\mathtt{v}_1\ \mathtt{v}_3\ \mathtt{v}_2)$
$(\mathtt{v}_3, \mathtt{v}_2)$	(distinct $\mathtt{v}_3$ $\mathtt{v}_2$) $\rightarrow$ [$\neg decision(\mathtt{v}_3) \vee decision(\mathtt{v}_2)$]	$(\mathtt{v}_1\ \mathtt{v}_3)$
$(\mathtt{v}_3, \mathtt{v}_3)$	(distinct $\mathtt{v}_3$ $\mathtt{v}_3$) $\rightarrow$ [$\neg decision(\mathtt{v}_3) \vee decision(\mathtt{v}_3)$]	none

Table 1: Correlation between symmetry and quantification for Φ_2 from (6)
Highlighted clauses represent the logical orbit $\varphi_2^{L(G)}$
none indicates the clause has no corresponding permutation $\gamma \in Sym(\mathtt{value}_3)$

We will first describe basic quantifier inference for protocols with independent sorts. This is done by analyzing the syntactic structure of each quantifier-free clause learned during incremental induction to derive a quantified form that expresses the clause's logical orbit. We later discuss extensions to this approach that consider protocols with dependent sorts, such as *ToyConsensus*, for which the basic single-clause quantifier inference may be insufficient.

5.1 Basic Quantifier Inference

Given a quantifier-free clause φ, quantifier inference seeks to derive a *compact* quantified predicate that *implicitly* represents, rather than explicitly enumerates, its logical orbit. The procedure must satisfy the following conditions:

Correctness – The inferred quantified predicate Φ should be logically-equivalent to the explicit logical orbit $\varphi^{L(G)}$.

Compactness – The number of quantified variables in Φ for each sort $\mathtt{s} \in S$ should be independent of the sort size $|\mathtt{s}|$. Intuitively, this condition ensures that the size of the quantified predicate, measured as the number of its quantifiers, remains bounded for *any* finite protocol instance, and more importantly, for the unbounded protocol.

SymIC3 constructs the orbit's quantified representation by a) inferring the required quantifiers for each sort separately, and b) stitching together the inferred quantifiers for the different sorts to form the final result. The key to capturing the logical orbit and deriving its compact quantified representation is a simple analysis of the *structural distribution* of each sort's constants in the target clause. Let $\pi(\varphi, \mathtt{s})$ be a partition of the constants of sort $\mathtt{s}$ in φ based on whether or not they appear *identically* in the literals of φ. Two constants $\mathtt{c_i}$ and $\mathtt{c_j}$ are identically-present in φ if they occur in φ and swapping them results in a logically-equivalent clause, i.e., $\varphi^{(\mathtt{c_i}\ \mathtt{c_j})} \equiv \varphi$. Let $\#(\varphi, \mathtt{s})$ be the number of constants of $\mathtt{s}$ that appear in φ, and let $|\pi(\varphi, \mathtt{s})|$ be the number of classes/cells in $\pi(\varphi, \mathtt{s})$. Consider the following scenarios for quantifier inference on sort $\mathtt{s}$:

A. $\#(\varphi, \mathtt{s}) < |\mathtt{s}|$ **(infer $\forall$)**

In this case, clause φ contains a strict subset of constants from sort $\mathtt{s}$, indicating that the number of literals in φ parameterized by $\mathtt{s}$ constants is *independent* of the sort size $|\mathtt{s}|$. Increasing sort size simply makes the orbit *longer* by adding more symmetrically-equivalent but logically-distinct clauses. An example of this case is φ_2 and Φ_2 in (6). The quantified predicate representing such an orbit requires $\#(\varphi, \mathtt{s})$ universally-quantified sort variables corresponding to the $\#(\varphi, \mathtt{s})$ sort constants in the clause, and expresses the orbit as an implication whose antecedent is a "distinct" constraint that ensures that the variables cannot be instantiated with identical constants.

B. $\#(\varphi, \mathtt{s}) = |\mathtt{s}|$

When all constants of a sort $\mathtt{s}$ appear in a clause, the above universal quantification yields a predicate with $|\mathtt{s}|$ quantified variables and fails the compactness requirement since the number of quantified variables becomes unbounded as the sort size increases. Correct quantification in this case must be inferred by examining the partition of the sort constants in the clause.

I. **Single-cell Partition i.e.,** $|\pi(\varphi, \mathtt{s})| = 1$ **(infer $\exists$)**

When all sort constants appear *identically* in φ, $\pi(\varphi, \mathtt{s})$ is a unit partition. Applying *any* permutation $\gamma \in Sym(\mathtt{s})$ to φ yields a logically-equivalent clause, i.e., the logical orbit in this case is just a single clause. Increasing the size of sort $\mathtt{s}$ simply yields a *wider* clause and suggests that such an orbit can be encoded as a predicate with a single existentially-quantified variable that ranges over all the sort constants. For example, the partition of the $\mathtt{value}_3$ sort constants in φ_1 from (4) is $\pi(\varphi_1, \mathtt{value}_3) = \{\{\mathtt{v}_1, \mathtt{v}_2, \mathtt{v}_3\}\}$ since all three constants appear identically in φ_1. The orbit of this clause is just itself and can be encoded as:

$$\Phi_1(\mathtt{value}_3) = \ \exists X \in \mathtt{value}_3 : \ vote(\mathtt{n}_1, X)$$

Also, since $\#(\varphi_1, \mathtt{node}_3) < |\mathtt{node}_3|$, universal quantification (as in Section 5.1.A) correctly captures the dependence of the clause's logical orbit on the $\mathtt{node}_3$ sort to get the overall quantified predicate Φ_1 in (7).

II. **Multi-cell Partition i.e.,** $|\pi(\varphi, \mathtt{s})| > 1$ **(infer $\forall\exists$)**

In this case, a fixed number of the constants of sort $\mathtt{s}$ appear differently in φ with the remaining constants appearing identically, resulting in a multi-cell partition. Specifically, assume that a number $0 < k < |\mathtt{s}|$ exists that is independent of $|\mathtt{s}|$ such that $\pi(\varphi, \mathtt{s})$ has $k + 1$ cells in which one cell has $|\mathtt{s}| - k$ identically-appearing constants and each of the remaining k cells contains one of the differently-appearing constants. It can be shown that the logical orbit in this case can be expressed by a quantified predicate with k universal quantifiers and a single existential quantifier. For example, the partition of the $\texttt{value}_3$ constants in the clause:

$$\varphi_4 = \neg decision(\texttt{v}_1) \vee decision(\texttt{v}_2) \vee decision(\texttt{v}_3)$$

is $\pi(\varphi_4, \texttt{value}_3) = \{\{\texttt{v}_1\}, \{\texttt{v}_2, \texttt{v}_3\}\}$ since $\texttt{v}_1$ appears differently from $\texttt{v}_2$ and $\texttt{v}_3$. The logical orbit of this clause is:

$$\begin{aligned} \varphi_4^{L(G)} = &[\ \neg decision(\texttt{v}_1) \vee decision(\texttt{v}_2) \vee decision(\texttt{v}_3)\ ] \wedge \\ &[\ \neg decision(\texttt{v}_2) \vee decision(\texttt{v}_1) \vee decision(\texttt{v}_3)\ ] \wedge \\ &[\ \neg decision(\texttt{v}_3) \vee decision(\texttt{v}_2) \vee decision(\texttt{v}_1)\ ] \end{aligned} \tag{9}$$

and can be compactly encoded with an outer universally-quantified variable corresponding to the sort constant in the singleton cell, and an inner existentially-quantified variable corresponding to the other $|\texttt{value}_3| - 1$ identically-present sort constants. A "distinct" constraint must also be conjoined with the literals involving the existentially-quantified variable to exclude the constant corresponding to the universally-quantified variable from the inner quantification. $\varphi_4^{L(G)}$ can thus be shown to be logically-equivalent to:

$$\Phi_4 = \forall Y \in \texttt{value}_3,\ \exists X \in \texttt{value}_3 :\ \neg decision(Y) \vee [(\text{distinct}\ \ Y\ \ X) \wedge decision(X)] \tag{10}$$

Combining Quantifier Inference for Different Sorts— The complete quantified predicate Φ representing the logical orbit of clause φ can be obtained by applying the above inference procedure to each sort in φ separately and in any order. This is possible since the sorts are assumed to be independent: the constants of one sort do not permute with the constants of a different sort. This will yield a predicate Φ that has the quantified prenex form $\forall^*\exists^* < \texttt{CNF expression} >$, where all universals for each sort are collected together and precede all the existential quantifiers.

It is interesting to note that this connection between symmetry and quantification suggests that an orbit can be visualized as a two-dimensional object whose height and width correspond, respectively, to the number of universally- and existentially-quantified variables. A proof of the correctness of this quantifier inference procedure can be found in Appendix B.2.

5.2 Quantifier Inference Beyond $\forall^*\exists^*$

We observed that for some protocols, particularly those that have dependent sorts such as *ToyConsensus*, the above inference procedure violates the compactness requirement. In other words, restricting inference to a $\forall^*\exists^*$ quantifier prefix causes the number of quantifiers to become unbounded as sort sizes increase. Recalling that the $\forall^*\exists^*$ pattern is inferred from the symmetries of a *single* clause, whose literals are the protocol's state variables, suggests that inference of more complex quantification patterns may necessitate that we examine the structural distribution of sort constants across *sets of clauses.* While this is an interesting possible direction for further exploration of the connection between symmetry and quantification, an alternative approach is to take advantage of the *formula structure* of the protocol's transition relation. For example, the transition relation of *ToyConsensus* is specified in terms of two quantified sub-formulas, *didNoteVote* and *chosenAt*, that can be viewed, in analogy with a sequential hardware circuit, as internal auxiliary non-state variables that act as "combinational" functions of the state variables. By allowing such auxiliary variables to appear explicitly in clauses learned during incremental induction, the quantified predicates representing the logical orbits of these clauses (according to the basic inference procedure in Section 5.1) will *implicitly* incorporate the quantifiers used in the auxiliary variable definitions and automatically have a quantifier prefix that generalizes the basic $\forall^*\exists^*$ template.

Revisiting ToyConsensus— When *SymIC3* is run on the finite instance *ToyConsensus*(3,3,3), it terminates with the following two strengthening assertions:

$$A_1 = \forall\ N \in \texttt{node}_3,\, V_1,\, V_2 \in \texttt{value}_3 :\ (\text{distinct}\ \ V_1\ \ V_2) \rightarrow \neg vote(N, V_1) \vee \neg vote(N, V_2) \quad (11)$$

$$\begin{aligned} A_2 &= \forall\ V \in \texttt{value}_3,\ \exists\ Q \in \texttt{quorum}_3.\ \neg decision(V) \vee chosenAt(Q, V) \quad (12) \\ &= \forall\ V \in \texttt{value}_3,\ \exists\ Q \in \texttt{quorum}_3.\ \neg decision(V) \vee [\ \forall\ N \in\ Q : vote(N, V)\ ] \end{aligned}$$

which, together with $\hat{P}$, serve as an inductive invariant proving that $\hat{P}$ holds for this instance. Both assertions are obtained using the basic quantifier inference procedure in Section 5.1 that produces a $\forall^*\exists^*$ quantifier prefix in terms of the clause variables. Note, however, that A_2 is expressed in terms of the auxiliary variable *chosenAt*. Substituting the definition of *chosenAt* yields an assertion with a $\forall\exists\forall$ quantifier prefix exclusively in terms of the protocol's state variables.

6 Finite Convergence Checks

Given a safe finite instance $\hat{\mathcal{P}} \triangleq \mathcal{P}(|\mathtt{s_1}|, \ldots, |\mathtt{s_n}|)$, let $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ denote the inductive invariant derived by *SymIC3* to prove that $\hat{P}$ holds in $\hat{\mathcal{P}}$. What remains is to determine the instance size $|\mathtt{s_1}|, \ldots, |\mathtt{s_n}|$ needed so that $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ is also an inductive invariant for all sizes. If the instance size is too small, $\hat{\mathcal{P}}$ may not include all protocol behaviors and $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ will not be inductive at larger sizes. As shown in the invisible invariant approach [9, 10, 58, 65, 70], increasing the instance size becomes necessary to include new protocol behaviors missing in $\hat{\mathcal{P}}$, until protocol behaviors *saturate*. We propose an *automatic* way to update the instance size and reach saturation by starting with an initial *base size* and iteratively increasing the size until *finite convergence* is achieved.

The initial base size can be chosen to be any non-trivial instance size and can be easily determined by a simple analysis of the protocol description. For example, any non-trivial instance of the *ToyConsensus* protocol should have $|\mathtt{node}| \geq 3$, $|\mathtt{quorum}| \geq 3$, and $|\mathtt{value}| \geq 2$.

Our finite convergence procedure can be seen as an integration of symmetry saturation and a stripped-down form of multi-dimensional mathematical induction, and has similarities with previous works on structural induction [35, 47] and proof convergence [25]. To determine if $Inv_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ is inductive for any size, the procedure performs the following checks for $1 \le i \le \mathtt{n}$:

$$\text{a) } \mathit{Init}(|\mathtt{s_1}|..\boxed{|\mathtt{s_i}|+1}..|\mathtt{s_n}|) \rightarrow \mathit{Inv}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}(|\mathtt{s_1}|..\boxed{|\mathtt{s_i}|+1}..|\mathtt{s_n}|) \quad (13)$$

$$\text{b) } \mathit{Inv}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}(|\mathtt{s_1}|..\boxed{|\mathtt{s_i}|+1}..|\mathtt{s_n}|) \wedge T(|\mathtt{s_1}|..\boxed{|\mathtt{s_i}|+1}..|\mathtt{s_n}|) \rightarrow \mathit{Inv}'_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}(|\mathtt{s_1}|..\boxed{|\mathtt{s_i}|+1}..|\mathtt{s_n}|) \quad (14)$$

where $\mathit{Inv}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}(|\mathtt{s_1}|..\boxed{|\mathtt{s_i}|+1}..|\mathtt{s_n}|)$ denotes the application of $\mathit{Inv}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ to an instance in which the size of sort $\mathtt{s_i}$ is increased by 1 while the sizes of the other sorts are unchanged.[3]

If all of these checks pass, we can conclude that $\mathit{Inv}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ is not specific to the instance size used to derive it and that we have reached *cutoff*, i.e., that $\mathit{Inv}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ is an inductive invariant for *any* size. Intuitively, this suggests that adding a new protocol component (e.g., client, server, node, proposer, acceptor) does not add any unseen unique behavior, and hence proving safety till the cutoff is sufficient to prove safety for any instance size. While we believe these checks are sufficient, we still do not have a formal convergence proof. In our implementation, we confirm convergence by performing the unbounded induction checks a) $\mathit{Init} \rightarrow \mathit{Inv}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$, and b) $\mathit{Inv}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|} \wedge T \rightarrow \mathit{Inv}'_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ noting that they may lie outside the decidable fragment of first-order logic.

On the other hand, failure of these checks, say for sort $\mathtt{s_i}$, implies that $\mathit{Inv}_{|\mathtt{s_1}|\ldots|\mathtt{s_n}|}$ will fail for larger sizes and cannot be inductive in the unbounded case, and we need to repeat *SymIC3* on a finite instance with an increased size for sort $\mathtt{s_i}$, i.e., $\hat{\mathcal{P}}_{new} \triangleq \mathcal{P}(|\mathtt{s_1}|,..,\boxed{|\mathtt{s_i}|+1},..,|\mathtt{s_n}|)$, to include new protocol behaviors that are missing in $\hat{\mathcal{P}}$.

Recall from (11) and (12), running *SymIC3* on *ToyConsensus*$(3,3,3)$ produces $\mathit{Inv}_{3,3,3} = A_1 \wedge A_2 \wedge \hat{P}$. $\mathit{Inv}_{3,3,3}$ passes checks (13) and (14) for instances *ToyConsensus*$(4,4,3)$ and *ToyConsensus*$(3,3,4)$, indicating finite convergence.[4] $\mathit{Inv}_{3,3,3}$ passes standard induction checks in the unbounded domain as well, establishing it as a proof certificate that proves the property as safe in *ToyConsensus*.

7 IC3PO: IC3 for Proving Protocol Properties

Given a protocol specification $\mathcal{P}$, IC3PO iteratively invokes *SymIC3* on finite instances of increasing size, starting with a given initial base size. Upon termination, IC3PO either a) reaches convergence on an inductive invariant $\mathit{Inv}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ that proves P for the unbounded protocol $\mathcal{P}$, or b) produces a counterexample trace $\mathit{Cex}_{|\mathtt{s_1}|,\ldots,|\mathtt{s_n}|}$ that serves as a finite witness to its violation in both the finite instance and the unbounded protocol. The detailed pseudo code of IC3PO is available in Appendix A.

We also explored a number of simple enhancements to IC3PO, such as strengthening the inferred quantified predicates whenever safely possible to do during incremental induction by a) dropping the "distinct" antecedent, and b) rearranging the quantifiers if the strengthened predicate is still unreachable from the previous frame. We describe these enhancements in Appendix C. The results presented in this paper were obtained without these enhancements.

[3] Sort dependencies, if any, should be considered when increasing a sort size.

[4] Since `quorum` is a dependent sort on `node`, it is increased together with the `node` sort.

Implementation— Our implementation of IC3PO is publicly available at https://github.com/aman-goel/ic3po. The implementation accepts protocol descriptions in the Ivy language [63] and uses the Ivy compiler to extract a quantified, logical formulation $\mathcal{P}$ in a customized VMT [22] format. We use a modified version [5] of the pySMT [34] library to implement our prototype, and use the Z3 [24] solver for all SMT queries. We use the SMT-LIB [14] theory of free sorts and function symbols with datatypes and quantifiers (`UFDT`), which allows formulating SMT queries for both, the finite and the unbounded domains. For a safe protocol, the inductive proof is printed in the Ivy format as an *independently check-able* proof certificate, which can be further validated with the Ivy verifier.

8 Evaluation

We evaluated IC3PO on a total of 29 distributed protocols including 4 problems from [53], 13 from [46], and 12 from [2]. This evaluation set includes fairly complex models of consensus algorithms as well as protocols such as two-phase commit, chord ring, hybrid reliable broadcast, etc. Several studies [16,32,42,46,53,63] have indicated the challenges involved in verifying these protocols.

All 29 protocols are safe based on manual verification. Even though finding counterexample traces is equally important, we limit our evaluation to safe protocols where the property holds, since inferring inductive invariants is the main bottleneck of existing techniques for verifying distributed protocols [30, 31, 63].

We compared IC3PO against the following 3 verifiers that implement state-of-the-art IC3-style techniques for automatic verification of distributed protocols:

– I4 [53] performs finite-domain IC3 (without accounting for symmetry) using the AVR model checker [39], followed by iteratively generalizing and checking the inductive invariant produced by AVR using Ivy.
– UPDR is the implementation of the PDR$^{\forall}$/UPDR algorithm [44] for verifying distributed protocols, from the *mypyvy* [4] framework.
– fol-ic3 [46] is a recent technique implemented in *mypyvy* that extends IC3 with the ability to infer inductive invariants with quantifier alternations.

All experiments were performed on an Intel (R) Xeon CPU (X5670). For each run, we used a timeout of 1 hour and a memory limit of 32 GB. All tools were executed in their respective default configurations. We used Z3 [24] version 4.8.9, Yices 2 [26] version 2.6.2, and CVC4 [13] version 1.7.

8.1 Results

Table 2 summarizes the experimental results. Apart from the number of problems solved, we compared the tools on 3 metrics: run time in seconds, proof size measured by the number of assertions in the inductive invariant for the unbounded protocol, and the total number of SMT queries made. Each tool uses SMT queries differently (e.g., I4 uses `QF_UF` for finite, `UF` for unbounded). Comparing the number of SMT queries still helps in understanding the run time behavior.

	Human		IC3PO			I4			UPDR			fol-ic3		
Protocol (#29)	Inv	info	Time	Inv	SMT	Time	Inv	SMT	Time	Inv	SMT	Time	Inv	SMT
tla-consensus	1		**0**	1	17	4	1	7	0	1	38	1	1	29
tla-tcommit	3		**1**	2	31	unknown		71	1	3	214	2	3	162
i4-lock-server	2		**1**	2	37	2	2	35	1	2	133	1	2	66
ex-quorum-leader-election	3		**3**	5	129	32	14	15429	11	3	1007	24	8	1078
pyv-toy-consensus-forall	4		**3**	4	105	unknown		5949	10	3	590	11	5	587
tla-simple	8		6	3	285	**4**	3	1319	timeout			timeout		
ex-lockserv-automaton	2		7	12	594	**3**	15	1731	21	9	3855	10	12	1181
tla-simpleregular	9		**8**	4	346	unknown		14787	timeout			57	9	314
pyv-sharded-kv	5		10	8	590	**4**	15	2101	6	7	784	22	10	522
pyv-lockserv	9		11	12	702	**3**	15	1606	14	9	3108	8	11	1044
tla-twophase	12		14	10	984	unknown		10505	67	14	12031	**9**	12	1635
i4-learning-switch	8		**14**	9	589	22	11	26345	timeout			timeout		
ex-simple-decentralized-lock	5		19	15	2219	14	22	5561	4	2	677	**4**	8	291
i4-two-phase-commit	11		27	11	2541	**4**	16	4045	16	9	2799	8	9	1083
pyv-consensus-wo-decide	5		**50**	9	1886	1144	42	41137	100	4	8563	168	26	5692
pyv-consensus-forall	7		**99**	10	3445	1006	44	156838	490	6	24947	2461	27	16182
pyv-learning-switch	8		**127**	13	3388	387	49	51021	278	11	3210	timeout		
i4-chord-ring-maintenance	18		**229**	12	6418	timeout			timeout			timeout		
pyv-sharded-kv-no-lost-keys	2	Æ	**3**	2	57	unknown		1232	unknown		73	3	2	51
ex-naive-consensus	4	Æ	**6**	4	239	unknown		15141	unknown		1325	73	18	414
pyv-client-server-ae	2	Æ $\triangleq$	**2**	2	49	unknown		1483	unknown		132	877	15	700
ex-simple-election	3	Æ $\triangleq$	**7**	4	268	unknown		2747	unknown		1147	32	10	222
pyv-toy-consensus-epr	4	Æ $\triangleq$	**9**	4	370	unknown		5944	unknown		473	70	14	217
ex-toy-consensus	3	Æ $\triangleq$	**10**	3	209	unknown		2797	unknown		348	21	8	124
pyv-client-server-db-ae	5	Æ $\triangleq$	**17**	6	868	unknown		81509	unknown		422	timeout		
pyv-hybrid-reliable-broadcast	8	Æ $\triangleq$	**587**	4	1474	unknown		34764	unknown		713	1360	23	3387
pyv-firewall	2	Æ $\leftrightarrows$	**2**	3	131	unknown		344	unknown		130	7	8	116
ex-majorityset-leader-election	5	Æ $\leftrightarrows$	**72**	7	1552	error			unknown		2350	timeout		
pyv-consensus-epr	7	Æ $\triangleq$ $\leftrightarrows$	**1300**	9	29601	unknown		177189	unknown		7559	1468	30	3355
No. of problems solved (out of 29)			**29**			13			14			23		
Uniquely solved			**3**			0			0			0		
For 10 cases solved by all: $\sum$ Time			**232**			2221			667			2711		
$\sum$ Inv			85			186			**52**			114		
$\sum$ SMT			**12160**			228490			45911			27168		

Table 2: Comparison of IC3PO against other state-of-the-art verifiers
Time: run time (seconds), Inv: # assertions in inductive proof, SMT: # SMT queries, Column "info" provides information on the strengthening assertions (i.e., A) in IC3PO's inductive proof: Æ indicates A has quantifier alternations, $\triangleq$ means A has definitions, and $\leftrightarrows$ means A adds quantifier-alternation cycles

IC3PO solved all 29 problems, while 10 protocols were solved by all the tools. The 5 rows at the bottom of Table 2 provide a summary of the comparison. Overall, compared to the other tools IC3PO is faster, requires fewer SMT queries, and produces shorter inductive proofs even for problems requiring inductive invariants with quantifier alternations (marked with Æ in Table 2).

We did a more extensive comparison between the two finite-domain incremental induction verifiers IC3PO and I4 (Appendix D), performed a statistical analysis using multiple runs with different solver seeds to account for the effect of randomness in SMT solving (Appendix E), compared the inductive proofs produced by IC3PO against human-written invariants (Appendix F), and performed a preliminary exploration of distributed protocols with totally-ordered domains and ring topologies (Appendix G).

8.2 Discussion

Comparing IC3PO and I4 clearly reveals the benefits of symmetric incremental induction. For example, I4 requires 7814 SMT queries to eliminate 443 CTIs when solving *ToyConsensus*(3,3,3), compared to 192 SMT calls and 13 CTIs for IC3PO. Even though both techniques perform finite incremental induction, symmetry-aware clause boosting in IC3PO leads to a factorial reduction in the number of SMT queries and yields compact inductive proofs.

Comparing IC3PO and UPDR reveals the benefits of finite-domain reasoning methods compared to direct unbounded verification. Even in cases where existential quantifier inference isn't necessary, symmetry-aware finite-domain reasoning gives IC3PO an edge both in terms of run time and the number of SMT queries.

Comparing IC3PO and fol-ic3, the only two verifiers that can infer invariants with a combination of universal and existential quantifiers, highlights the advantage of IC3PO's approach over the separators-based technique [46] used in fol-ic3. The significant performance edge that IC3PO has over fol-ic3 is due to the fact that a) reasoning in IC3PO is primarily in a (small) finite domain compared to fol-ic3's unbounded reasoning, and b) unlike fol-ic3 which enumeratively searches for specific quantifier patterns, IC3PO finds the required invariants without search by automatically inferring their patterns from the symmetry of the protocol.

Overall, the evaluation confirms the main hypothesis of this paper, that it is possible to use the relationship between symmetry and quantification to scale the verification of distributed protocols beyond the current state-of-the-art.

9 Related Work

Introduced by Lamport, TLA+ is a widely-used language for the specification and verification of distributed protocols [15,59]. The accompanying TLC model checker can perform automatic verification on a finite instance of a TLA+ specification, and can also be configured to employ symmetry to improve scalability. However, TLC is primarily intended as a debugging tool for small finite instances and not as a tool for inferring inductive invariants.

Several manual or semi-automatic verification techniques (e.g., using interactive theorem proving or compositional verification) have been proposed for deriving system-level proofs [21, 36, 42, 43, 62, 69]. These techniques generally require a deep understanding of the protocol being verified and significant manual effort to guide proof development. The Ivy [63] system improves on these techniques by graphically displaying CTIs and interactively asking the user to provide strengthening assertions that can eliminate them.

Verification of parameterized systems using SMT solvers is further explored in MCMT [67], Cubicle [23], and paraVerifier [52]. Abdulla et al. [7] proposed *view abstraction* to compute the reachable set for finite instances using forward reachability until cutoff is reached. Our technique builds on these works with the capability to automatically infer the required quantified inductive invariant using the latest advancements in model checking, by combining symmetry-aware clause learning and quantifier inference in finite-domain incremental induction. The use of derived/ghost variables has been recognized as important in [48, 58, 61]. IC3PO utilizes protocol structure, namely auxiliary definitions in the protocol specification, to automatically infer inductive invariants with complex quantifier alternations.

Several recent approaches (e.g., UPDR [45], QUIC3 [41], Phase-UPDR [32], fol-ic3 [46]) extend IC3/PDR to automatically infer quantified inductive invariants.

Unlike IC3PO, these techniques rely heavily on unbounded SMT solving.

Our work is closest in spirit to FORHULL-N [25] and I4 [53, 54]. Similar to IC3PO, these techniques perform incremental induction over small finite instances of a parameterized system and employ a generalization procedure that transforms finite-domain proofs to quantified inductive invariants that hold for all parameter values. Dooley and Somenzi proposed FORHULL-N to verify parameterized reactive systems by running bit-level IC3 and generalizing the learnt clauses into candidate universally-quantified proofs through a process of proof saturation and convex hull computation. These candidate proofs involve modular linear arithmetic constraints as antecedents in a way such that they approximate the protocol behavior beyond the current finite instance, and their correctness is validated by checking them until the cutoff is reached. I4 uses an ad hoc generalization procedure to obtain universally-quantified proofs from the finite-domain inductive invariants generated by the AVR model checker [39].

10 Conclusions and Future Work

IC3PO is, to our knowledge, the first verification system that uses the synergistic relationship between symmetry and quantification to automatically infer the quantified inductive invariants required to prove the safety of symmetric protocols. Recognizing that symmetry and quantification are alternative ways of capturing invariance, IC3PO extends the incremental induction algorithm to learn clause orbits, and encodes these orbits with corresponding logically-equivalent and compact quantified predicates. IC3PO employs a systematic procedure to check for finite convergence, and outputs quantified inductive invariants, with both universal and existential quantifiers, that hold for all protocol parameters. Our evaluation demonstrates that IC3PO significantly is a significant improvement over the current state-of-the-art.

Future work includes exploring methods to utilize the regularity in totally-ordered domains during reachability analysis, investigating techniques to counter undecidability in practical distributed systems verification, and exploring enhancements to further improve the scalability to complex distributed protocols and their implementations. As a long-term goal, we aim towards automatically inferring inductive invariants for complicated distributed protocols, such as Paxos [50, 51], by building further on this initial work.

Data Availability Statement and Acknowledgments

The software and data sets generated and analyzed during the current study, including all experimental data, evaluation scripts, and IC3PO source code are available at https://github.com/aman-goel/nfm2021exp. We thank the developers of pySMT [34], Z3 [24], and Ivy [63] for making their tools openly available. We thank the authors of the I4 project [53] for their help in shaping some of the ideas presented in this paper.

References

1. Client server protocol in ivy. http://microsoft.github.io/ivy/examples/client_server_example.html
2. A collection of distributed protocol verification problems. https://github.com/aman-goel/ivybench
3. The ivy language and verifier. http://microsoft.github.io/ivy
4. mypyvy (github). https://github.com/wilcoxjay/mypyvy
5. pySMT: A library for SMT formulae manipulation and solving. https://github.com/aman-goel/pysmt
6. Toy consensus protocol. https://github.com/microsoft/ivy/blob/master/examples/ivy/toy_consensus.ivy
7. Abdulla, P., Haziza, F., Holík, L.: Parameterized verification through view abstraction. International Journal on Software Tools for Technology Transfer **18**(5), 495–516 (2016)
8. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
9. Arons, T., Pnueli, A., Ruah, S., Xu, Y., Zuck, L.: Parameterized verification with automatically computed inductive assertions? In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification. pp. 221–234. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
10. Balaban, I., Fang, Y., Pnueli, A., Zuck, L.D.: Iiv: An invisible invariant verifier. In: International Conference on Computer Aided Verification. pp. 408–412. Springer (2005)
11. Balyo, T., Froleyks, N., Heule, M.J., Iser, M., Järvisalo, M., Suda, M.: Proceedings of sat competition 2020: Solver and benchmark descriptions (2020)
12. Barner, S., Grumberg, O.: Combining symmetry reduction and under-approximation for symbolic model checking. In: International Conference on Computer Aided Verification. pp. 93–106. Springer (2002)
13. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11). Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (Jul 2011), http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf, snowbird, Utah
14. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
15. Beers, R.: Pre-RTL formal verification: an intel experience. In: Proceedings of the 45th annual Design Automation Conference. pp. 806–811 (2008)
16. Berkovits, I., Lazic, M., Losa, G., Padon, O., Shoham, S.: Verification of threshold-based distributed algorithms by decomposition to decidable logics. CoRR **abs/1905.07805** (2019), http://arxiv.org/abs/1905.07805

17. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of parameterized verification. Synthesis Lectures on Distributed Computing Theory **6**(1), 1–170 (2015). https://doi.org/10.2200/S00658ED1V01Y201508DCT013
18. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation. pp. 70–87. VMCAI'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=1946284.1946291
19. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. In: Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science. pp. 428–439 (1990)
20. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model checking: 10^{20} States and Beyond. Information and Computation **98**(2), 142–170 (1992)
21. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the tla+ proof system. In: International Joint Conference on Automated Reasoning. pp. 142–148. Springer (2010)
22. Cimatti, A., Roveri, M., Griggio, A., Irfan, A.: Verification Modulo Theories. http://www.vmt-lib.org (2011)
23. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: A parallel smt-based model checker for parameterized systems. In: International Conference on Computer Aided Verification. pp. 718–724. Springer (2012)
24. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
25. Dooley, M., Somenzi, F.: Proving parameterized systems safe by generalizing clausal proofs of small instances. In: International Conference on Computer Aided Verification. pp. 292–309. Springer (2016)
26. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 737–744. Springer International Publishing, Cham (2014)
27. Een, N., Mishchenko, A., Brayton, R.: Efficient Implementation of Property Directed Reachability. In: Formal Methods in Computer Aided Design (FMCAD'11). pp. 125 – 134 (Oct 2011)
28. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: International conference on theory and applications of satisfiability testing. pp. 502–518. Springer (2003)
29. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal methods in system design **9**(1-2), 105–131 (1996)
30. Feldman, Y.M.Y., Sagiv, M., Shoham, S., Wilcox, J.R.: Learning the boundary of inductive invariants. CoRR **abs/2008.09909** (2020), https://arxiv.org/abs/2008.09909
31. Feldman, Y.M., Immerman, N., Sagiv, M., Shoham, S.: Complexity and information in invariant inference. Proceedings of the ACM on Programming Languages **4**(POPL), 1–29 (2019)
32. Feldman, Y.M., Wilcox, J.R., Shoham, S., Sagiv, M.: Inferring inductive invariants from phase structures. In: International Conference on Computer Aided Verification. pp. 405–425. Springer (2019)
33. Fraleigh, J.B.: A First Course in Abstract Algebra. Addison Wesley Longman, Reading, Massachusetts, 6th edn. (2000)
34. Gario, M., Micheli, A.: Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In: SMT workshop. vol. 2015 (2015)
35. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. Journal of the ACM (JACM) **39**(3), 675–735 (1992)

36. v. Gleissenthall, K., Kıcı, R.G., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony: synchronous verification of asynchronous distributed programs. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
37. Godefroid, P.: Exploiting symmetry when model-checking software. In: Formal Methods for Protocol Engineering and Distributed Systems, pp. 257–275. Springer (1999)
38. Goel, A., Sakallah, K.: Model checking of verilog rtl using ic3 with syntax-guided abstraction. In: NASA Formal Methods Symposium. pp. 166–185. Springer (2019)
39. Goel, A., Sakallah, K.: Avr: Abstractly verifying reachability. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 413–422. Springer (2020)
40. Goel, A., Sakallah, K.A.: Empirical Evaluation of IC3-Based Model Checking Techniques on Verilog RTL Designs. In: Proc. of the Design, Automation and Test in Europe Conference (DATE). pp. 618–621. Florence, Italy (March 2019)
41. Gurfinkel, A., Shoham, S., Vizel, Y.: Quantifiers on demand. In: International Symposium on Automated Technology for Verification and Analysis. pp. 248–266. Springer (2018)
42. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 1–17. ACM (2015)
43. Hoenicke, J., Majumdar, R., Podelski, A.: Thread modularity at many levels: a pearl in compositional verification. ACM SIGPLAN Notices **52**(1), 473–485 (2017)
44. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. J. ACM **64**(1) (Mar 2017). https://doi.org/10.1145/3022187, https://doi.org/10.1145/3022187
45. Karbyshev, A., Bjørner, N., Itzhaky, S., Rinetzky, N., Shoham, S.: Property-directed inference of universal invariants or proving their absence. Journal of the ACM (JACM) **64**(1), 1–33 (2017)
46. Koenig, J.R., Padon, O., Immerman, N., Aiken, A.: First-order quantified separators. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 703–717. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385412.3386018, https://doi.org/10.1145/3385412.3386018
47. Kurshan, R.P., McMillan, K.: A structural induction theorem for processes. In: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing. pp. 239–247 (1989)
48. Lamport, L.: Proving the correctness of multiprocess programs. IEEE transactions on software engineering (2), 125–143 (1977)
49. Lamport, L.: Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
50. Lamport, L.: The part-time parliament. In: Concurrency: the Works of Leslie Lamport, pp. 277–317 (2019)
51. Lamport, L., et al.: Paxos made simple. ACM Sigact News **32**(4), 18–25 (2001)
52. Li, Y., Pang, J., Lv, Y., Fan, D., Cao, S., Duan, K.: Paraverifier: An automatic framework for proving parameterized cache coherence protocols. In: International Symposium on Automated Technology for Verification and Analysis. pp. 207–213. Springer (2015)
53. Ma, H., Goel, A., Jeannin, J.B., Kapritsos, M., Kasikci, B., Sakallah, K.A.: I4: Incremental inference of inductive invariants for verification of distributed protocols.

In: Proceedings of the 27th Symposium on Operating Systems Principles. ACM (2019)
54. Ma, H., Goel, A., Jeannin, J.B., Kapritsos, M., Kasikci, B., Sakallah, K.A.: Towards automatic inference of inductive invariants. In: Proceedings of the Workshop on Hot Topics in Operating Systems. pp. 30–36. ACM (2019)
55. Marques-Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (1999)
56. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Norwell, MA, USA (1993)
57. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC. pp. 530–535 (2001)
58. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 299–313. Springer (2007)
59. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Communications of the ACM **58**(4), 66–73 (2015)
60. Norris IP, C., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design **9**(1), 41–75 (Aug 1996). https://doi.org/10.1007/BF00625968, https://doi.org/10.1007/BF00625968
61. Owicki, S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Communications of the ACM **19**(5), 279–285 (1976)
62. Owre, S., Rushby, J.M., Shankar, N.: Pvs: A prototype verification system. In: International Conference on Automated Deduction. pp. 748–752. Springer (1992)
63. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 614–630. PLDI '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908118, http://doi.acm.org/10.1145/2908080.2908118
64. Piskac, R., de Moura, L., Bjørner, N.: Deciding effectively propositional logic using dpll and substitution sets. Journal of Automated Reasoning **44**(4), 401–424 (Apr 2010). https://doi.org/10.1007/s10817-009-9161-6, https://doi.org/10.1007/s10817-009-9161-6
65. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 82–97. Springer (2001)
66. Pong, F., Dubois, M.: A new approach for the verification of cache coherence protocols. IEEE Transactions on Parallel and Distributed Systems **6**(8), 773–787 (1995)
67. Ranise, S., Ghilardi, S.: Backward reachability of array-based systems by smt solving: Termination and invariant synthesis. Logical Methods in Computer Science **6** (2010)
68. Sistla, A.P., Gyuris, V., Emerson, E.A.: Smc: a symmetry-based model checker for verification of safety and liveness properties. ACM Transactions on Software Engineering and Methodology (TOSEM) **9**(2), 133–166 (2000)
69. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 357–368. PLDI '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737958, http://doi.acm.org/10.1145/2737924.2737958

70. Zuck, L., Pnueli, A.: Model checking and abstraction to the aid of parameterized systems (a survey). Computer Languages, Systems & Structures **30**(3-4), 139–169 (2004)

Appendices

We include additional/supplementary material in the appendices, as follows:

Appendix A: *IC3PO Pseudo Code (detailed)*
- Presents the detailed pseudo code of IC3PO and *SymIC3*

Appendix B: *Proof of Correctness*
- Provides a correctness proof for symmetry-aware clause boosting during incremental induction (Section 4), and a correctness proof for quantifier inference (Section 5)

Appendix C: *Simple Enhancements to the SymIC3 Algorithm*
- Describes simple enhancements to *SymIC3* learning as briefly mentioned in Section 7

Appendix D: *Effect of Symmetry Learning in Incremental Induction*
- Evaluates the effect of symmetry-aware learning in finite-domain incremental induction with a detailed comparison between IC3PO and I4

Appendix E: *Statistical Analysis with Multiple SMT Solver Seeds*
- Provides a statistical analysis of the experiments from Section 8 through multiple runs for each tool with different solver seeds

Appendix F: *Comparison against Human-Written Invariants*
- Compares IC3PO's automatically-generated quantified inductive invariants against human-written invariant proofs on several metrics

Appendix G: *Ordered Domains, Ring Topology, and Special Variables*
- Describes an extension to IC3PO that allows handling totally-ordered domains, as well as further details relating to ring topology and special variables, along with a preliminary evaluation

Appendix H: *Finite Instance Sizes used in the Experiments*
- Lists down the instance sizes for IC3PO and I4 for each protocol in the evaluation (Section 8)

Appendix A IC3PO Pseudo Code (detailed)

This section presents the detailed pseudo code of IC3PO and *SymIC3*.

1 **procedure** *IC3PO*($\mathcal{P}$, σ_0) - - $\mathcal{P} \triangleq [S, R, \mathit{Init}, T, P]$, and σ_0 is the initial base size
2   *reuse* $\leftarrow \{\}$
3     $\sigma \leftarrow \sigma_0$
4   *Inv*, *Cex* $\leftarrow$ *SymIC3*($\hat{\mathcal{P}}$, *reuse*) - - run symmetric incremental induction on $\hat{\mathcal{P}} \triangleq \mathcal{P}(\sigma)$
5   **if** *Cex* is not empty **then** - - counterexample found
6   **return** `Violated`, *Cex* - - property is violated
7   **else** - - property proved for the finite protocol instance $\hat{\mathcal{P}}$
8   **for each** $\mathtt{s_i} \in S$ **do**
9   **if not** *IsInductiveInvariantFinite*(*Inv*, $\mathcal{P}(\sigma^{+}[\mathtt{s_i}])$) **then**
10   *reuse* $\leftarrow \{\ \Phi \mid \Phi \in \mathit{Inv}$ and $\mathit{Init} \rightarrow \Phi$ and $\mathit{Init} \wedge T \rightarrow \Phi'$ in $\mathcal{P}(\sigma^{+}[\mathtt{s_i}])\ \}$
11                 $\sigma \leftarrow \sigma^{+}[\mathtt{s_i}]$ - - failed convergence checks for sort $\mathtt{s_i}$, increase instance size
12   **go to** Line 4 - - re-run *SymIC3* with the increased size
13   **if not** *IsInductiveInvariantUnbounded*(*Inv*, $\mathcal{P}$) **then**
14   **< never occurred >** - - unbounded check failed
15   **return** `Error`, `Increase` σ_0
16   **return** `Safe`, *Inv* - - property is proved safe with proof certificate *Inv*

Algorithm 1: ***IC3** for **P**roving **P**rotocol **P**roperties*

Algorithm 1 presents the detailed pseudo code of IC3PO. Let $\sigma : S \rightarrow \mathbb{N}$ be a function that maps each sort $\mathtt{s_i} \in S$ to a sort size $|\mathtt{s_i}|$. Given a protocol specification $\mathcal{P}$ and an initial base size σ_0, IC3PO invokes *SymIC3* on the finite protocol instance $\hat{\mathcal{P}} \triangleq \mathcal{P}(\sigma)$, where σ is initialized to σ_0 (lines 2-4). Upon termination, *SymIC3* either a) produces a quantified inductive invariant *Inv* that proves the property for $\hat{\mathcal{P}}$, or b) a counterexample trace *Cex* that serves as a finite witness to its violation in both $\hat{\mathcal{P}}$ and the unbounded protocol $\mathcal{P}$ (lines 4-6). If the property holds for $\hat{\mathcal{P}}$, IC3PO performs finite convergence checks (Section 6) to check whether or not the invariant extends beyond $\hat{\mathcal{P}}$ (lines 8-12), by checking whether or not *Inv* is an inductive invariant for the larger finite instance $\hat{\mathcal{P}}^{\mathtt{i}} \triangleq \mathcal{P}(\sigma^{+}[\mathtt{s_i}])$ for each $\mathtt{s_i} \in S$, where $\sigma^{+}[\mathtt{s_i}] \triangleq [\sigma \quad \text{EXCEPT} \ ![\mathtt{s_i}] = \sigma(\mathtt{s_i}) + 1]$. If all finite checks pass, *Inv* is checked whether an inductive invariant in the unbounded domain (lines 13-15) using the standard induction checks– a) $\mathit{Init} \rightarrow \mathit{Inv}$, and b) $\mathit{Inv} \wedge T \rightarrow \mathit{Inv}'$ in the unbounded domain. If all these checks pass, IC3PO emits the unbounded invariant *Inv*, that holds for the unbounded $\mathcal{P}$ and is a proof certificate for the safety property (line 16). Otherwise, it re-starts *SymIC3* on a finite instance with an increased size $\sigma^{+}[\mathtt{s_i}]$ (lines 11-12), while seeding in all the strengthening assertions in *Inv* that are safe to learn in the first frame for the new *SymIC3* iteration (line 10).

Algorithm 2 describes the symmetric incremental induction algorithm. The procedure first checks whether the property can be trivially violated (lines 19-22), and if not, starts recursively deriving and blocking counterexamples-to-induction (CTI) from the topmost frame (lines 24-35). Given a solver model $\mathtt{m}$,

17 **procedure** *SymIC3*($\hat{\mathcal{P}}$, *reuse*) - - $\hat{\mathcal{P}} \triangleq [S, R, \hat{Init}, \hat{T}, \hat{P}]$
- - *reuse* is a set of seed assertions that are safe to learn in the frame F_1
18 $F \leftarrow \emptyset$, $Cex \leftarrow \emptyset$ - - $\hat{\mathcal{P}}$, F, Cex are global data structures
19 **if** `SAT ? [` $\hat{Init} \wedge \neg\hat{P}$ `]: model m` **then** - - initial states check
20 $state \leftarrow StateAsCube(\mathtt{m})$ - - get a single state from model `m`, in cube form
21 $Cex.extend(state)$ - - property is trivially violated
22 **return** $\emptyset$, Cex - - return the counterexample
23 $F.extend(\hat{Init})$ - - setup the initial frame
24 **while** $\top$ **do**
25 $N \leftarrow F.size() - 1$
26 **if** `SAT ? [` $F_N \wedge \hat{T} \wedge \neg\hat{P}'$ `]: model m` **then**
- - check the topmost frame for counterexample-to-induction (CTI)
27 $state \leftarrow StateAsCube(\mathtt{m})$ - - found a CTI
28 **if** $SymRecBlockCube(state,\ N)$ **then** - - try recursively blocking the CTI
29 **return** $\emptyset$, Cex - - failed to block CTI, return the counterexample
30 **else** - - no CTI in the topmost frame
31 $F.extend(\hat{P})$ - - add a new frame
32 **if** $N = 0$ **then** - - add reusable seed assertions to the frame F_1
33 $F[1].add(reuse)$
34 **if** *ForwardPropagate()* **then** - - propagate inductive assertions forward
35 **return** $F_{converged}$, $\emptyset$
- - frames converged, return $F_{converged}$ as the inductive invariant

36 **procedure** $SymRecBlockCube(cti, i)$ - - cti can reach $\neg\hat{P}$ in $F.size() - i$ steps
37 $Cex.extend(cti)$ - - add the CTI to the counterexample
38 **if** $i = 0$ **then** - - check if reached the initial states
39 **return** $\top$ - - reached initial states, property is violated
40 **if** `SAT ? [` $F_{i-1} \wedge \hat{T} \wedge cti'$ `]: model m` **then**
- - check if cti is reachable from previous frame
41 $state \leftarrow StateAsCube(\mathtt{m})$
- - $state$ is the new CTI reachable to $\neg\hat{P}$ in $(F.size() - i) + 1$ steps
42 **return** $SymRecBlockCube(state,\ i - 1)$ - - try blocking the new CTI
43 **else** - - cti is unreachable from the previous frame
44 $uc' \leftarrow MinimalUnsatCore(F_{i-1} \wedge \hat{T},\ cti')$ - - get MUS from UNSAT query
45 $\varphi \leftarrow \neg uc$ - - negate uc to get the quantifier-free clause
46 $\Phi \leftarrow SymBoost\forall\exists(\varphi)$ - - symmetry-aware clause boosting with quantifier inference
47 $\Phi \leftarrow AntecedentReduction(\Phi,\ i)$ - - antecedent reduction (optional), Appendix C.1
48 $\Phi \leftarrow EprReduction(\Phi,\ i)$ - - EPR reduction (optional), Appendix C.2
49 `Learn(`Φ`, `F_i`)` - - learn Φ in frame i
50 **return** $\bot$
51

Algorithm 2: *Symmetric Incremental Induction*

a state cube is derived as a single state represented as a cube, i.e., a conjunction of literals assigning each state variable with a value based on its assignment in `m` (lines 20, 27, 41). Lines 32-33 add the seed assertions in the given *reuse* set to the first frame F_1. *SymIC3* differs from the standard IC3 algorithm majorly

52 **procedure** *SymBoost*$\forall\exists(\varphi)$ - - φ is the quantifier-free clause
53 $\quad V_\forall \leftarrow \{\},\ V_\exists \leftarrow \{\}$ - - a set of universally/existential quantified variables
54 $\quad$ *body* $\leftarrow \varphi$ - - starting with φ, *body* is recursively generated
- - $V_\forall$, $V_\exists$ and *body* are global data structures
55 $\quad$ **for each** sort $\mathtt{s}$ that appears in clause φ **do**
56 $\qquad \pi(\varphi, \mathtt{s}) \leftarrow$ *PartitionDistribution*$(\varphi, \mathtt{s})$
- - create a partition on constants in $\mathtt{s}$ based on their occurrence in φ
57 $\qquad$ **if** $\#(\varphi, \mathtt{s}) < |\mathtt{s}|$ **then**
58 $\qquad\quad (V_\forall, V_\exists, body) \leftarrow$ *Infer*$\forall(\varphi, \pi(\varphi, \mathtt{s}))$ - - infer $\forall$ for sort $\mathtt{s}$, refer §5.1.A
59 $\qquad$ **else if** $|\pi(\varphi, \mathtt{s})| = 1$ **then** - - partition $\pi(\varphi, \mathtt{s})$ contains a single cell
60 $\qquad\quad (V_\forall, V_\exists, body) \leftarrow$ *Infer*$\exists(\varphi, \pi(\varphi, \mathtt{s}))$ - - infer $\exists$ for sort $\mathtt{s}$, refer §5.1.B.I
61 $\qquad$ **else if** *all but a few scenario* **then** - - partition $\pi(\varphi, \mathtt{s})$ contains multiple cells
62 $\qquad\quad (V_\forall, V_\exists, body) \leftarrow$ *Infer*$\forall\exists(\varphi, \pi(\varphi, \mathtt{s}))$ - - infer $\forall\exists$ for sort $\mathtt{s}$, refer §5.1.B.II
63 $\qquad$ **else**
64 $\qquad\quad$ `< never occurred >`
- - infer $\forall$ by default (may not be compact, though correct for the current instance)
65 $\qquad\quad (V_\forall, V_\exists, body) \leftarrow$ *Infer*$\forall(\varphi, \pi(\varphi, \mathtt{s}))$
66 $\quad \Phi \leftarrow \forall V_\forall.\ \exists V_\exists.\ body$ - - stitch quantifiers for different sorts as $\forall_{\dots}\ \exists_{\dots}\ < body >$
67 $\quad$ **return** Φ - - Φ is the quantified predicate to learn in a *SymIC3* frame

Algorithm 3: *Symmetry-aware Clause Boosting with Quantifier Inference*

in symmetry-aware quantified learning (line 46) and simple enhancements (lines 47-48).

The core of the *SymIC3* algorithm is the *SymBoost*$\forall\exists$ algorithm, presented in Algorithm 3. *SymBoost*$\forall\exists$ is a simple and extendable procedure to perform symmetry-aware clause boosting and quantifier inference, as explained in detail in Sections 4 and 5. Starting from a given quantifier-free clause φ, the algorithm constructs a symmetrically-boosted quantified predicate Φ (line 67) by iteratively inferring quantifiers for each sort $\mathtt{s}$ (lines 55-65), and stitching them together (line 66). The algorithm maintains a set of universal and existential variables (line 53) and a *body* (line 54), that are iteratively modified based on the quantifier inference for each sort. For each sort $\mathtt{s}$, the algorithm first generates $\pi(\varphi, \mathtt{s})$ (line 56) based on how constants in sort $\mathtt{s}$ appear in the literals of φ (whether identically or not). The next step is to infer quantifiers using $\#(\varphi, \mathtt{s})$ and $\pi(\varphi, \mathtt{s})$ (lines 57-65): a) infer universal quantifiers when $\#(\varphi, \mathtt{s}) < |\mathtt{s}|$, b) otherwise if all constants of $\mathtt{s}$ appear in φ identically, infer existential quantifier, c) otherwise if *all but a few scenario*, infer $\forall\exists$ based on the partitioning of constants in $\pi(\varphi, s)$, and d) otherwise, infer $\forall$ by default (this case has not occurred). Changing the iteration order in line 55 doesn't result in any difference, and is ensured during the recursive building of the *body*. At the end, a single quantified predicate Φ is derived by stitching together the quantified variables in $V_\forall$ and $V_\exists$ with the *body* as $\forall_{\dots}\ \exists_{\dots}\ < body >$ (line 66).

Appendix B Proof of Correctness

Appendix B.1 Correctness Proof for Symmetric Incremental Induction

This section provides a correctness proof for symmetry-aware clause boosting during incremental induction (Section 4).

Like the invariance of $\hat{Init}$, $\hat{T}$, and $\hat{P}$ under any permutation $\gamma \in G$ (refer (2)), the logical orbit of a clause φ is also invariant under such permutations, i.e.,

$$\left[\varphi^{L(G)}\right]^{\gamma} \leftrightarrow \varphi^{L(G)}$$

Lemma 1. *For any SymIC3 frame F_i, $F_i^{\gamma} \equiv F_i$ for any $\gamma \in G$.*

Proof. Recall that $\hat{Init}^{\gamma} \equiv \hat{Init}$ and $\hat{P}^{\gamma} \equiv \hat{P}$. The condition $F_i^{\gamma} \equiv F_i$ is trivially true for $i = 0$ since $F_0 = \hat{Init}$. When $i > 0$, the condition is true during frame initialization since each frame is initialized to $\hat{P}$. When blocking a cube $\neg\varphi$ in F_i, incremental induction with symmetry boosting refines F_i with the complete logical orbit $\varphi^{L(G)}$ of φ. Since $\left[\varphi^{L(G)}\right]^{\gamma} \equiv \varphi^{L(G)}$, the logical invariance of F_i under γ, continues to be preserved in all backward reachability updates. $\square$

The following theorem establishes the correctness of symmetry-aware clause boosting in incremental induction.

Theorem 1. *If a quantifier-free cube $\neg\varphi$ is unreachable from frame F_{i-1}, i.e., $F_{i-1} \wedge \hat{T} \wedge \neg[\varphi]'$ is unsatisfiable, then $F_{i-1} \wedge \hat{T} \wedge \neg[\varphi^{L(G)}]'$ is also unsatisfiable.*

Proof. Let $Q \triangleq F_{i-1} \wedge \hat{T} \wedge \neg[\varphi]'$ and assume that Q is unsatisfiable. Consider any permutation $\gamma \in G$ and the corresponding permuted formula $Q^{\gamma} \triangleq F_{i-1}^{\gamma} \wedge \hat{T}^{\gamma} \wedge \neg\left[\varphi^{\gamma}\right]'$. Since permuting the sort constants simply re-arranges the protocol's state variables in a formula without affecting its satisfiability, Q and Q^{γ} must be equisatisfiable, and hence Q^{γ} is unsatisfiable.

Noting that $\hat{T}$ and F_{i-1} are invariant under $\gamma \in G$ (from (2) and Lemma 1), we obtain $Q^{\gamma} = F_{i-1} \wedge \hat{T} \wedge \neg\left[\varphi^{\gamma}\right]'$ proving that if cube $\neg\varphi$ is unreachable from frame F_{i-1}, then its image under any $\gamma \in G$ is also unreachable. Therefore, $F_{i-1} \wedge \hat{T} \wedge \neg[\varphi^{L(G)}]'$ is unsatisfiable. $\square$

Appendix B.2 Correctness Proof for Quantifier Inference

This section provides a correctness proof sketch for quantifier inference (Section 5).

Theorem 2. *Given a finite instance $\hat{\mathcal{P}}$, let φ be such that $0 < \#(\varphi, \mathtt{s}) < |\mathtt{s}|$ for some sort $\mathtt{s} \in S$. Let $\Phi(\mathtt{s})$ be the quantified predicate obtained by applying SymIC3's quantifier inference for $\mathtt{s}$. $\Phi(\mathtt{s})$ is logically equivalent to $\varphi^{L(Sym(\mathtt{s}))}$.*

Proof. Let γ be any permutation in $\mathit{Sym}(\mathtt{s})$, and let $n \triangleq \#(\varphi, \mathtt{s})$. Let $\widehat{\varphi}$ be the clause obtained by replacing in φ each constant $\mathtt{c_i} \in \mathtt{s}$ by a corresponding variable V_i of sort $\mathtt{s}$.

Let $A \triangleq [\,(V_1 = \mathtt{c_1}) \wedge \cdots \wedge (V_n = \mathtt{c_n})\,] \rightarrow \widehat{\varphi}$. By the transitivity of equality, $A \equiv \varphi$. Let $B \triangleq \bigwedge_{\gamma \,\in\, \mathit{Sym}(\mathtt{s})} A^{\gamma}$. Since $A \equiv \varphi$, therefore, $B \equiv \varphi^{L(\mathit{Sym}(\mathtt{s}))}$, and can be re-written as:

$$B = \bigwedge_{\gamma \,\in\, \mathit{Sym}(\mathtt{s})} (\,[\,(V_1 = \mathtt{c_1}) \wedge \cdots \wedge (V_n = \mathtt{c_n})\,] \rightarrow \widehat{\varphi}\,)^{\gamma} \tag{15}$$

$$= \bigwedge_{\gamma \,\in\, \mathit{Sym}(\mathtt{s})} [\,(V_1 = \mathtt{c_1}) \wedge \cdots \wedge (V_n = \mathtt{c_n})\,]^{\gamma} \rightarrow \widehat{\varphi} \tag{16}$$

$$= \forall\ V_1 \ldots V_n.\ (\text{distinct}\ V_1 \ldots V_n) \rightarrow \widehat{\varphi} \tag{17}$$

$$= \Phi(\mathtt{s}) \tag{18}$$

(15) & (16) are equal since $\widehat{\varphi}$ does not contain any constant of sort $\mathtt{s}$, and hence $[\widehat{\varphi}]^{\gamma} \equiv \widehat{\varphi}$. (16) & (17) are equal since the antecedents in (16) cover all possible assignments of variables $(V_1, \ldots, V_n)$ to n distinct constants of sort $\mathtt{s}$. There are total $\binom{|\mathtt{s}|}{n} \times n!$ possible assignments of the variables in (17) to n distinct constants of sort $\mathtt{s}$, one each corresponding to the $\binom{|\mathtt{s}|}{n} \times n!$ permutations in $\mathit{Sym}(\mathtt{s})$ that yield a logically-distinct antecedent in (16). (17) & (18) are equal since given $\#(\varphi, \mathtt{s}) < |\mathtt{s}|$.
Since $B \equiv \varphi^{L(\mathit{Sym}(\mathtt{s}))}$, therefore $\Phi(\mathtt{s}) \equiv \varphi^{L(\mathit{Sym}(\mathtt{s}))}$. □

Theorem 3. *Given a finite instance $\hat{\mathcal{P}}$, let φ be such that all constants of a sort $\mathtt{s} \in S$ appear identically in the literals of φ. Let $\Phi(\mathtt{s})$ be the quantified predicate obtained by applying SymIC3's quantifier inference for $\mathtt{s}$. $\Phi(\mathtt{s})$ is logically equivalent to $\varphi^{L(\mathit{Sym}(\mathtt{s}))}$.*

Proof. Let γ be any permutation in $\mathit{Sym}(\mathtt{s})$. Since given all constants in sort $\mathtt{s}$ appear identically in the literals of φ, therefore $\pi(\varphi, \mathtt{s})$ consists of a single cell, and any permutation $\gamma \in \mathit{Sym}(\mathtt{s})$ does not result in a new logically-distinct clause, i.e., $\varphi^{\gamma} \equiv \varphi$. As a result, $\varphi^{L(\mathit{Sym}(\mathtt{s}))} \equiv \varphi$.
Without loss of generality, φ can be written as:

$$\varphi = \varphi_{others} \vee \bigvee_{\mathtt{c_i} \,\in\, \mathtt{s}} \varphi_{\mathtt{s}}(\mathtt{c_i}) \tag{19}$$

where φ_{others} is the disjunction of literals in φ that do not contain any constant of sort $\mathtt{s}$, and $\varphi_{\mathtt{s}}(\mathtt{c_i})$ is the disjunction of literals in φ that contain a constant $\mathtt{c_i} \in \mathtt{s}$. Note that φ_{others} can be $\bot$.

Let $\widehat{\varphi_{\mathtt{s}}}$ be the clause obtained by replacing in $\varphi_{\mathtt{s}}(\mathtt{c_i})$ each constant $\mathtt{c_i} \in \mathtt{s}$ by a variable V of sort $\mathtt{s}$. Note that since all constants of sort $\mathtt{s}$ appear identically in the literals of φ, therefore $\widehat{\varphi_{\mathtt{s}}}$ is the same for each $\mathtt{c_i} \in \mathtt{s}$. The clause φ can therefore be re-written as:

$$\varphi = \varphi_{others} \vee \bigvee_{\mathtt{c_i}\, \in\, \mathtt{s}} (V = \mathtt{c_i}) \rightarrow \widehat{\varphi_{\mathtt{s}}} \tag{20}$$

$$= \varphi_{others} \vee \exists\ V.\ \ \widehat{\varphi_{\mathtt{s}}} \tag{21}$$

$$= \Phi(\mathtt{s}) \tag{22}$$

(19) & (20) are equal due to the transitivity of equality. (20) & (21) are equal since expanding the existential quantifier as a disjunction over all possible assignments of the variable V gives the expression in (20). (21) & (22) are equal since $\#(\varphi, \mathtt{s}) = |\mathtt{s}|$ and $|\pi(\varphi, \mathtt{s}) = 1|$, and hence *SymIC3* infers $\Phi(\mathtt{s})$ as (21). Since $\varphi \equiv \varphi^{L(Sym(\mathtt{s}))}$, therefore $\Phi(\mathtt{s}) \equiv \varphi^{L(Sym(\mathtt{s}))}$. □

Appendix C Simple Enhancements to the IC3PO Algorithm

This section describes simple enhancements to *SymIC3* learning as mentioned in Section 7.

Appendix C.1 Antecedent Reduction

Antecedent reduction strengthens a quantified predicate Φ by dropping the antecedent (distinct ...) and checking the unsatisfiability of the query [$F_{i-1} \wedge \hat{T} \wedge \neg\Phi'$]. For example, Φ_2 from (6) can possibly be strengthened by dropping (distinct X_1 X_2) from the antecedent to get Φ_{new}, if the query [$F_{i-1} \wedge \hat{T} \wedge \neg\Phi'_{new}$] is unsatisfiable, where

$$\Phi_{new} = \ \forall X_1, X_2 \in \texttt{value}.\ \neg decision(X_1) \vee decision(X_2)$$

If instead, the query is satisfiable, the original predicate Φ_2 should be learnt.

Appendix C.2 EPR Reduction

With the quantifier inference employed by *SymBoost*$\forall\exists$ (Algorithm 3), *SymIC3* can produce predicates with alternating quantifiers, which can result in quantifier-alternation cycles. For example, our running example already includes a quantifier alternation from `quorum` $\longrightarrow$ `node` (Figure 1, line 3). Consider an example predicate:

$$\Phi = \ \forall Y \in \texttt{node},\ \exists Z \in \texttt{quorum}.\ member(Y, Z)$$

The quantified predicate Φ adds the arc `node` $\longrightarrow$ `quorum`, generating a quantifier-alternation cycle:

$$\texttt{quorum} \longrightarrow \texttt{node} \longrightarrow \texttt{quorum}$$

Even though there are no undecidability concerns while reasoning over the finite instance $\hat{\mathcal{P}}$ (since the sort domains are finite), it is desirable to avoid quantifier-alternation cycles and derive the invariant in the EPR fragment [64] of FOL. Restricting to the EPR fragment allows robustly checking the inductive invariant over the unbounded protocol $\mathcal{P}$. Note that IC3PO performs invariant construction as well as finite convergence checks both in a finite domain (as detailed in Section 7).

We can additionally strengthen the learning to be within the EPR fragment, by *pushing out* existential quantifiers and avoid generation of quantifier-alternation cycle. For example, the EPR-reduced version Φ_{epr} of Φ is

$$\Phi_{epr} = \ \exists Z \in \texttt{quorum},\ \forall Y \in \texttt{node}.\ member(Y, Z)$$

If we consider both Φ and its negation $\neg\Phi$ (as needed during induction checks), EPR-reduction basically *flips* the quantifier-alternation arcs. For example, the quantifier-alternation graph with the EPR-reduced predicate Φ_{epr} (instead of Φ) is:

$$\texttt{quorum} \longrightarrow \texttt{node} \longleftarrow \texttt{quorum}$$

$\neg\Phi_{epr}$ adds the arc `node` $\longleftarrow$ `quorum`.

Logically, *pushing out* the existential quantifier results in a *reduced/stricter* formula, with $\Phi_{epr} \rightarrow \Phi$, but $\Phi \not\rightarrow \Phi_{epr}$ (hence we call it EPR "reduction"). Intuitively, this difference is analogous to the difference in the statements:

$Likes_{\forall\exists}$:= Everyone likes someone $\quad$ $Likes_{\exists\forall}$:= Someone is liked by everyone

where $Likes_{\exists\forall} \rightarrow Likes_{\forall\exists}$, but $Likes_{\forall\exists} \not\rightarrow Likes_{\exists\forall}$.

We can add EPR reduction in the incremental induction procedure with *SymIC3*, that enables learning the EPR-reduced form Φ_{epr} instead of Φ *only when it is safe*, i.e., only when $\neg\Phi_{epr}$ is still unreachable from the previous incremental induction frame F_{i-1}. We do so by checking the unsatisfiability of the finite domain (and hence decidable) query [$F_{i-1} \wedge \hat{T} \wedge \neg\Phi'_{epr}$]. If the query is unsatisfiable, we learn the strengthened EPR-reduced predicate Φ_{epr}. Else, the original form, i.e., Φ, is learnt.

Note- Both simple enhancements presented in this section were left disabled in IC3PO for all experiments in this paper to focus the evaluation on the main paper contents. Initial investigation with these enhancements shows significant benefits in performance and robustness, with hardly any overhead.

Appendix D Effect of Symmetry Learning in Incremental Induction

This section evaluates the effect of symmetry-aware clause boosting in finite-domain incremental induction with a detailed comparison between IC3PO and I4.

Table 3 compares the effect of symmetry-aware learning in incremental induction for the problems solved by both IC3PO and I4. The table compares the number of SMT solver calls made and counterexamples-to-induction (CTI) encountered during the incremental induction procedure, as well as the number of assertions in the final (quantified) inductive invariant. *SymIC3*'s symmetry boosting helps IC3PO to make orders of magnitude fewer SMT solver calls compared to I4 and solve the problem after discovering many fewer CTIs.

Overall, Table 3 justifies the runtime speedups observed in Table 2, and confirms the benefits of symmetry-aware learning.

	IC3PO			I4		
Protocol (#13)	#SMT	#CTI	#Inv	#SMT	#CTI	#Inv
tla-consensus	13	0	1	7	0	1
i4-lock-server	31	1	2	35	2	2
ex-quorum-leader-election	117	7	5	15429	847	14
tla-simple	273	23	3	1319	41	3
ex-lockserv-automaton	568	51	12	1731	156	15
pyv-sharded-kv	572	25	8	2101	170	15
pyv-lockserv	676	58	12	1606	142	15
i4-learning-switch	567	32	9	26345	1310	11
ex-simple-decentralized-lock	2155	87	15	5561	490	22
i4-two-phase-commit	2131	68	11	4045	288	16
pyv-consensus-wo-decide	1866	141	9	41137	2451	42
pyv-consensus-forall	3423	247	10	156838	10316	44
pyv-learning-switch	3352	112	13	51021	3639	49
$\sum$ #SMT	15744	(**19.5x** better)		307175		
$\sum$ #CTI	852	(**23.3x** better)		19852		
$\sum$ #Inv	110	(**2.3x** better)		249		

Table 3: Comparison of different incremental induction metrics between IC3PO and I4 for the problems solved by both
#SMT: number of solver queries, #CTI: number of counterexamples-to-induction
#Inv: number of assertions in the final (quantified) inductive invariant

Appendix E Statistical Analysis with Multiple SMT Solver Seeds

This section provides a statistical analysis of the experiments from Section 8 through multiple runs for each tool with different solver seeds.

Different tools perform best with different SMT solvers (e.g., I4 uses a combination of Yices 2 [26] and Z3 [24], fol-ic3 uses Z3 and CVC4 [13], while UPDR and IC3PO use Z3).[5] For the results presented in Table 2, a fixed SMT solver seed (i.e., *seed* = 1) was used for all tools. To get an idea of the effect of randomness in SMT solving, we performed 10 runs with different solver seeds for each tool on all protocols, and compared the runtime mean and standard deviation.

	IC3PO			I4			UPDR			fol-ic3		
Protocol (#29)	#	Time	σ	#	Time	σ	#	Time	σ	#	Time	σ
tla-consensus	✓	0	0	✓	5	0	✓	**0**	0	✓	1	0
tla-tcommit	✓	**1**	0	✗			✓	1	0	✓	2	0
i4-lock-server	✓	**1**	0	✓	2	0	✓	1	0	✓	1	0
ex-quorum-leader-election	✓	**3**	0	✓	32	0	✓	10	1	✓	21	3
pyv-toy-consensus-forall	✓	**3**	1	✗			✓	6	1	✓	11	1
tla-simple	✓	34	93	✓	5	0	✗			2	**3**	0
ex-lockserv-automaton	✓	9	3	✓	**3**	0	✓	21	1	✓	11	0
tla-simpleregular	✓	**8**	4	✗			✗			✓	79	22
pyv-sharded-kv	✓	8	1	✓	**4**	0	✓	6	0	✓	22	0
pyv-lockserv	✓	11	4	✓	**3**	0	✓	15	2	✓	8	0
tla-twophase	✓	**15**	3	✗			✓	99	12	✓	16	8
i4-learning-switch	✓	**20**	8	✓	22	0	✗			✗		
ex-simple-decentralized-lock	✓	20	0	✓	14	0	✓	**4**	0	✓	4	0
i4-two-phase-commit	✓	79	167	✓	**4**	0	✓	19	3	✓	9	0
pyv-consensus-wo-decide	✓	**40**	9	✓	1226	37	✓	107	16	✓	82	45
pyv-consensus-forall	✓	**135**	72	✓	1042	36	✓	398	86	✓	2277	553
pyv-learning-switch	✓	**161**	66	✓	387	17	✓	209	56	1	311	0
i4-chord-ring-maintenance	8	**1289**	1191	✗			✗			✗		
pyv-sharded-kv-no-lost-keys	✓	**2**	0	✗			✗			✓	5	1
ex-naive-consensus	✓	**5**	1	✗			✗			✓	80	17
pyv-client-server-ae	✓	**1**	0	✗			✗			✓	630	130
ex-simple-election	✓	172	522	✗			✗			✓	**38**	8
pyv-toy-consensus-epr	✓	**14**	8	✗			✗			✓	47	12
ex-toy-consensus	✓	**11**	5	✗			✗			✓	22	4
pyv-client-server-db-ae	✓	**32**	30	✗			✗			✗		
pyv-hybrid-reliable-broadcast	6	**157**	211	✗			✗			6	2264	740
pyv-firewall	✓	**2**	0	✗			✗			✓	6	1
ex-majorityset-leader-election	✓	**63**	47	✗			✗			✗		
pyv-consensus-epr	2	1968	943	✗			✗			5	**768**	404
No. of problems solved (out of 29)		**29**			13			14			25	
Uniquely solved		**3**			0			0			0	
For 11 cases solved by all: $\sum$ Time		**470**			2727			795			2752	

Table 4: Statistical comparison of IC3PO against other state-of-the-art verifiers #: number of runs where successfully solved (out of 10) (✓ means 10, ✗ means 0), Time: runtime mean (in seconds), σ: runtime standard deviation (in seconds)

[5] We used Yices 2 version 2.6.2, Z3 version 4.8.9 and CVC4 version 1.7.

Appendix F Comparison against Human-Written Invariants

Figure 2 compares IC3PO's automatically-generated inductive invariants against the human-written proofs on several metrics. Our evaluation shows IC3PO produces compact proofs of sizes comparable to the manually-written inductive invariants, even shorter than the human proofs on several occasions. As a side benefit, IC3PO's inductive invariants are pretty-printed in the Ivy format [3], and thus, can also be independently checked/validated through Ivy.

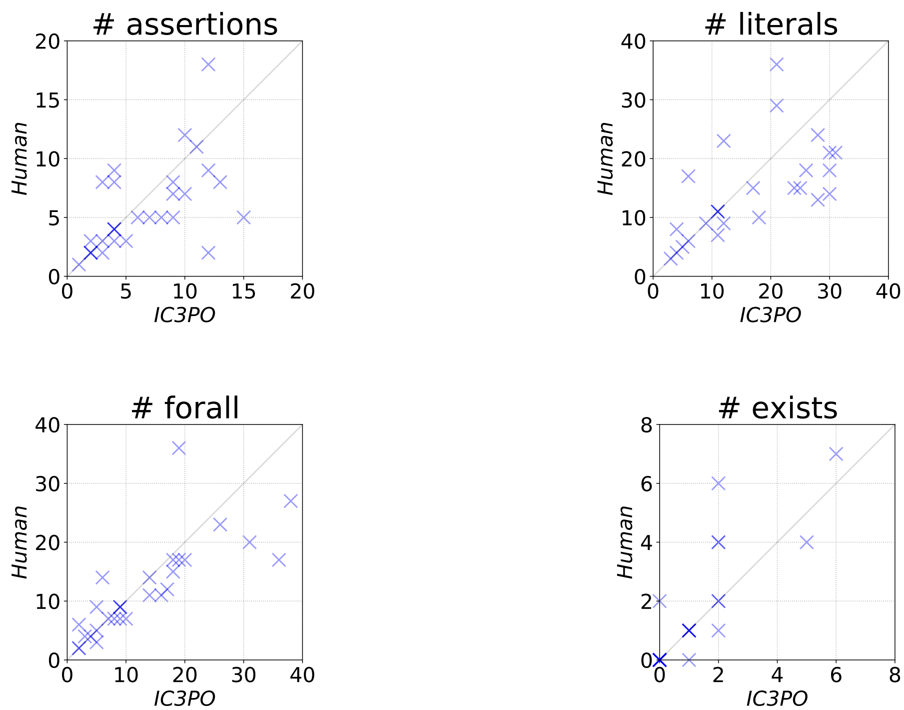

Fig. 2: Comparison of IC3PO's inductive invariant against *human-written* proof IC3PO is on x-axis, *human-written* on y-axis

Appendix G Ordered Domains, Ring Topology and Special Variables

This section describes an extension to IC3PO that allows handling totally-ordered domains, as well as further details relating to ring topology and special variables (along with a preliminary evaluation).

		Human	IC3PO			I4			UPDR			fol-ic3		
Protocol (#13)		Inv	Time	Inv	SMT	Time	Inv	SMT	Time	Inv	SMT	Time	Inv	SMT
ex-distributed-lock-abstract	$<$	12	**15**	11	946	timeout			timeout			timeout		
ex-decentralized-lock	$<$	4	**25**	5	654	288	32	104616	timeout			timeout		
ex-distributed-lock-maxheld	$<$	6	**58**	10	1866	422	73	100749	timeout			3210	48	4557
pyv-ticket	$<$	14	**65**	8	1896	error			228	13	15936	98	26	3177
i4-database-chain-replication	$<$	9	98	6	1382	**20**	10	6111	timeout			1222	16	5455
ex-decentralized-lock-abstract	$<$	6	**126**	18	5069	error			timeout			timeout		
i4-distributed-lock	$<$	7	**155**	10	3472	3280	102	410364	timeout			1191	64	4875
ex-ring-not-dead	$\circlearrowright$ $<$	2	10	2	161	unknown		3327	unknown		28	**6**	3	100
ex-ring	$\circlearrowright$ $<$	3	11	3	269	**6**	9	678	9	2	662	7	3	248
ex-ring-id-not-dead-limited	$\circlearrowright$ $<$	2	24	2	250	unknown		29083	unknown		31	**7**	3	81
pyv-ring-id-not-dead	$\circlearrowright$ $<$	2	37	2	275	unknown		182325	unknown		31	**8**	3	86
pyv-ring-id	$\circlearrowright$ $<$	4	73	4	869	420	11	225789	99	3	4107	**28**	9	594
i4-leader-election-in-ring	$\circlearrowright$ $<$	6	323	5	2907	749	25	359776	114	3	4229	**59**	17	1378
No. of problems solved	(out of 13)			**13**			7			4			10	
Uniquely solved				**2**			0			0			0	
For 3 cases solved by all:	$\sum$ Time			407			1176			224			**95**	
	$\sum$ Inv			12			45			**8**			29	
	$\sum$ SMT			4045			586243			8998			**2220**	

Table 5: Comparison of IC3PO against other state-of-the-art verifiers
Time: runtime in seconds, Inv: # assertions in the inductive invariant,
SMT: # SMT solver queries made, $\circlearrowright$ indicates protocol has a ring topology, $<$ indicates protocol has a totally-ordered domain

Ordered domains like *epoch*, *time*, etc. are not symmetric, which makes such domains unsuitable to directly apply a symmetry argument. Specifically, restricting an unbounded ordered domain to a finite size results in introducing boundary cases with a "max" element, complicating finite-domain behavior.

Even in the presence of ordered domains, symmetry-aware learning can still be applied to all the un-ordered domains while leaving the ordered domains as unbounded. As an initial exploration, we devised a hybrid procedure in IC3PO where ordered domains are handled in an unbounded fashion, in the same manner as in UPDR, while all other domains are handled in the *SymIC3*-style symmetry-aware and finite manner. We use UPDR's *diagram-based abstraction* to infer quantifiers for the ordered domain, while using *SymBoost*$\forall\exists$ (Algorithm 3) for the un-ordered domains.[6]

For the protocols that involve a ring topology, a ring domain, generally composed of identical components arranged in a ring topology, retains domain symmetry since the position of each individual component in the ring is left uninitialized and can be arbitrarily permuted. Hence, *SymIC3* can be directly

[6] We refer the reader to [44] for a complete description of incremental induction with diagram-based abstraction.

applied. The same is true for protocols that have special components, like a special *start_node* that initially holds the lock in a distributed lock. Non-Boolean functions and variables are modeled in relational form with equality predicates. For example, permuting the predicate ($start\_node = \mathtt{n}_1$) with the permutation ($\mathtt{n}_1\ \mathtt{n}_2$) gives the permuted predicate ($start\_node = \mathtt{n}_2$). IC3PO exploits the symmetry in the sort domains, not symmetries over the protocol symbols (i.e., relations, functions and variables), and hence is unaffected by the presence of special protocol symbols.

Table 5 summarizes the experimental results for 13 protocols with totally-ordered domains, collected again from [2, 46, 53]. IC3PO solves all 13 problems and shows the advantages of symmetry-aware learning even when applied only to a subset of protocol's domains. We believe additional exploration is needed for these cases, where the non-symmetric regularity in totally-ordered domains can be further utilized to improve learning during incremental induction.

Appendix H Finite Instance Sizes used in Experiments

Table 6 lists down the initial base instance sizes used for IC3PO runs in the evaluation (Section 8) for each protocol. The table also includes the final *cutoff* instance sizes reached, where the corresponding *Inv* generalizes/saturates to be an inductive proof for any size. Note again that IC3PO updates the instance sizes automatically, as described in Section 6.

Protocol		Finite instance sizes used for IC3PO
tla-consensus		$\texttt{value} = 2$
tla-tcommit		$\texttt{resource-manager} = 2$
i4-lock-server		$\texttt{client} = 2,\ \texttt{server} = 1$
ex-quorum-leader-election	E	$\texttt{node} = 2 \mapsto 3,\ \texttt{nset} = 2$
pyv-toy-consensus-forall	E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
tla-simple	$\circlearrowright\ E$	$\texttt{node} = 2,\ \texttt{pcstate} = 3,\ \texttt{value} = 2 \mapsto 3$
ex-lockserv-automaton		$\texttt{node} = 2$
tla-simpleregular	$\circlearrowright\ E$	$\texttt{node} = 2,\ \texttt{pcstate} = 4,\ \texttt{value} = 2 \mapsto 3$
pyv-sharded-kv		$\texttt{key} = 2,\ \texttt{node} = 2,\ \texttt{value} = 2$
pyv-lockserv		$\texttt{node} = 2$
tla-twophase		$\texttt{resource-manager} = 2$
i4-learning-switch		$\texttt{node} = 2 \mapsto 3,\ \texttt{packet} = 1$
ex-simple-decentralized-lock		$\texttt{node} = 2 \mapsto 4$
i4-two-phase-commit		$\texttt{node} = 4$
pyv-consensus-wo-decide	E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3$
pyv-consensus-forall	E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
pyv-learning-switch	E	$\texttt{node} = 2 \mapsto 4$
i4-chord-ring-maintenance	$\circlearrowright\ E$	$\texttt{node} = 3 \mapsto 5$
pyv-sharded-kv-no-lost-keys	E	$\texttt{key} = 2,\ \texttt{node} = 2,\ \texttt{value} = 2$
ex-naive-consensus	E	$\texttt{node} = 3,\ \texttt{quorum} = 3,\ \texttt{value} = 3$
pyv-client-server-ae	E	$\texttt{node} = 2,\ \texttt{request} = 2 \mapsto 3,\ \texttt{response} = 2$
ex-simple-election	E	$\texttt{acceptor} = 2 \mapsto 3,\ \texttt{proposer} = 2,\ \texttt{quorum} = 1 \mapsto 3$
pyv-toy-consensus-epr	E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
ex-toy-consensus	E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
pyv-client-server-db-ae	E	$\texttt{db-request-id} = 2 \mapsto 3,\ \texttt{node} = 2,\ \texttt{request} = 2 \mapsto 3,\ \texttt{response} = 2$
pyv-hybrid-reliable-broadcast	E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum-a} = 2 \mapsto 3,\ \texttt{quorum-b} = 2$
pyv-firewall	E	$\texttt{node} = 2 \mapsto 3$
ex-majorityset-leader-election	E	$\texttt{node} = 2 \mapsto 3,\ \texttt{nodeset} = 2 \mapsto 3$
pyv-consensus-epr	E	$\texttt{node} = 2 \mapsto 3,\ \texttt{quorum} = 1 \mapsto 3,\ \texttt{value} = 2$
ex-distributed-lock-abstract	$<$	$\texttt{epoch} = \infty,\ \texttt{node} = 2$
ex-decentralized-lock	$<$	$\texttt{node} = 2,\ \texttt{time} = \infty$
ex-distributed-lock-maxheld	$<$	$\texttt{epoch} = \infty,\ \texttt{node} = 2$
pyv-ticket	$<$	$\texttt{thread} = 2 \mapsto 3,\ \texttt{ticket} = \infty$
i4-database-chain-replication	$E\ <$	$\texttt{key} = 1,\ \texttt{node} = 2,\ \texttt{operation} = 2 \mapsto 3,\ \texttt{transaction} = \infty$
ex-decentralized-lock-abstract	$<$	$\texttt{node} = 2 \mapsto 4,\ \texttt{time} = \infty$
i4-distributed-lock	$<$	$\texttt{epoch} = \infty,\ \texttt{node} = 2$
ex-ring-not-dead	$\circlearrowright\ E\ <$	$\texttt{node} = 3$
ex-ring	$\circlearrowright\ <$	$\texttt{node} = 3$
ex-ring-id-not-dead-limited	$\circlearrowright\ E\ <$	$\texttt{id} = 3,\ \texttt{node} = 3$
pyv-ring-id-not-dead	$\circlearrowright\ E\ <$	$\texttt{id} = \infty,\ \texttt{node} = 3$
pyv-ring-id	$\circlearrowright\ <$	$\texttt{id} = \infty,\ \texttt{node} = 3$
i4-leader-election-in-ring	$\circlearrowright\ <$	$\texttt{id} = \infty,\ \texttt{node} = 3$

Table 6: Finite instance sizes used for IC3PO

$\texttt{s} = x$ denotes sort $\texttt{s}$ has both initial base size and final cutoff size x

$\texttt{s} = x \mapsto y$ denotes sort $\texttt{s}$ has initial size x and final cutoff size y (incrementally increased by IC3PO automatically)

$\texttt{s} = \infty$ denote the totally-ordered sort $\texttt{s}$ is left unbounded

$\circlearrowright$ indicates protocol has a ring topology, $<$ indicates protocol has an ordered domain

E indicates the protocol description has $\exists$

Table 7 lists down the instance sizes used for I4 runs in the evaluation (Section 8) for each protocol.

Protocol		Finite instance sizes used for I4
tla-consensus		value = 2
tla-tcommit		resource-manager = 2
i4-lock-server		client = 2, server = 1
ex-quorum-leader-election	E	node = 3, nset = 3
pyv-toy-consensus-forall	E	node = 3, quorum = 3, value = 2
tla-simple	$\circlearrowright$ E	node = 3, pcstate = 3, value = 3
ex-lockserv-automaton		node = 2
tla-simpleregular	$\circlearrowright$ E	node = 3, pcstate = 4, value = 3
pyv-sharded-kv		key = 2, node = 2, value = 2
pyv-lockserv		node = 2
tla-twophase		resource-manager = 3
i4-learning-switch		node = 3, packet = 2
ex-simple-decentralized-lock		node = 4
i4-two-phase-commit		node = 5
pyv-consensus-wo-decide	E	node = 3, quorum = 3
pyv-consensus-forall	E	node = 3, quorum = 3, value = 2
pyv-learning-switch	E	node = 4
i4-chord-ring-maintenance	$\circlearrowright$ E	node = 4
pyv-sharded-kv-no-lost-keys	E	key = 3, node = 3, value = 3
ex-naive-consensus	E	node = 3, quorum = 3, value = 3
pyv-client-server-ae	E	node = 3, request = 3, response = 3
ex-simple-election	E	acceptor = 3, proposer = 2, quorum = 3
pyv-toy-consensus-epr	E	node = 3, quorum = 3, value = 2
ex-toy-consensus	E	node = 3, quorum = 3, value = 2
pyv-client-server-db-ae	E	db-request-id = 3, node = 3, request = 3, response = 3
pyv-hybrid-reliable-broadcast	E	node = 3, quorum-a = 3, quorum-b = 3
pyv-firewall	E	node = 3
ex-majorityset-leader-election	E	node = 3, nodeset = 3
pyv-consensus-epr	E	node = 3, quorum = 3, value = 2
ex-distributed-lock-abstract	$<$	epoch = 4, node = 2
ex-decentralized-lock	$<$	node = 2, time = 4
ex-distributed-lock-maxheld	$<$	epoch = 4, node = 2
pyv-ticket	$<$	thread = 3, ticket = 5
i4-database-chain-replication	E $<$	key = 1, node = 2, operation = 3, transaction = 3
ex-decentralized-lock-abstract	$<$	node = 4, time = 4
i4-distributed-lock	$<$	epoch = 4, node = 2
ex-ring-not-dead	$\circlearrowright$ E $<$	node = 3
ex-ring	$\circlearrowright$ $<$	node = 3
ex-ring-id-not-dead-limited	$\circlearrowright$ E $<$	id = 3, node = 3
pyv-ring-id-not-dead	$\circlearrowright$ E $<$	id = 4, node = 3
pyv-ring-id	$\circlearrowright$ $<$	id = 4, node = 3
i4-leader-election-in-ring	$\circlearrowright$ $<$	id = 4, node = 3

Table 7: Finite instance sizes used for I4

$\circlearrowright$ indicates protocol has a ring topology, $<$ indicates protocol has an ordered domain

E indicates the protocol description has $\exists$