DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols

Jianan Yao
Columbia University

Runzhou Tao
Columbia University

Ronghui Gu
Columbia University

Jason Nieh
Columbia University

Abstract

Distributed systems are complex and difficult to build correctly. Formal verification can provably rule out bugs in such systems, but finding an inductive invariant that implies the safety property of the system is often the hardest part of the proof. We present DuoAI, an automated system that quickly finds inductive invariants for verifying distributed protocols by reducing SMT query costs in checking invariants with existential quantifiers. DuoAI enumerates the strongest candidate invariants that hold on validate states from protocol simulations, then applies two methods in parallel, returning the result from the method that succeeds first. One checks all candidate invariants and weakens them as needed until it finds an inductive invariant that implies the safety property. Another checks invariants without existential quantifiers to find an inductive invariant without the safety property, then adds candidate invariants with existential quantifiers to strengthen it until the safety property holds. Both methods are guaranteed to find an inductive invariant that proves desired safety properties, if one exists, but the first reduces SMT query costs when more candidate invariants with existential quantifiers are needed, while the second reduces SMT query costs when few candidate invariants with existential quantifiers suffice. We show that DuoAI verifies more than two dozen common distributed protocols automatically, including various versions of Paxos, and outperforms alternative methods both in the number of protocols it verifies and the speed at which it does so, including solving Paxos more than two orders of magnitude faster than previous methods.

1 Introduction

The world relies on distributed systems, but these systems are increasingly complex and hard to design and implement correctly. To address this problem, developers are starting to turn to formal verification techniques to prove the correctness of distributed systems [11, 20, 35]. This involves formally verifying that desired safety properties hold for the distributed protocol. A safety property is an invariant that should hold true at any point in a system's execution. It ensures the protocol does not reach invalid or dangerous states. For example, the safety property for a distributed lock protocol [11] is that no two nodes in the system hold a lock at the same time. The proof requires finding an invariant that implies the safety property, then proving that it is inductive. An invariant is inductive if it holds for all initial states of the system, and is preserved on all valid transitions so that it holds for any reachable state of the system.

Unfortunately, finding an inductive invariant is often the hardest part of the proof [21]. Invariants can be expressed as logical formulas consisting of universal ($\forall$) and existential ($\exists$) quantifiers with a certain number of variables, and a set of logical relations among the variables. Recent work has focused on automating the process of finding an inductive invariant, but has various limitations. I4 [21] was the first to automate the process, but provides no guarantee that it can find the inductive invariant and does not work for invariants with existential quantifiers. Our previous work DistAI [38] provides speed advantages over I4 and a guarantee of finding an $\exists$-free inductive invariant if one exists, but also does not work for invariants with existential quantifiers. FOL-IC3 [13] was the first to handle existential quantifiers, but is inefficient due to its heavy use of expensive SMT queries. It often fails to find invariants for protocols that can be solved by other approaches such as I4 and DistAI. SWISS [10] can successfully find an inductive invariant for Paxos, but does not work for more complex protocols such as stoppable Paxos [27]. It fails or is much slower than I4 and DistAI for many protocols without existential quantifiers.

We present DuoAI, an automated system to quickly find inductive invariants for verifying distributed protocols, with and without existential quantifiers, including complex versions of Paxos. Even though a distributed protocol may be used in very large systems, its invariants are likely to be concise, as protocols need to be designed and understood by humans to be correct. As a result, DuoAI operates in formula space and considers smaller formulas first to enumerate candidate invariants, which are then checked by an SMT solver. Formula size is defined by a maximum number of quantified variables (a variable and its quantifier $\forall$ or $\exists$) and relations. If DuoAI does

not succeed with smaller formulas, it increases the formula size and repeats the process until an inductive invariant is found. Although the formula space within a given formula size is finite, checking all possible invariants for even a modest size formula is prohibitively expensive, especially since SMT solvers are particularly inefficient at checking invariants with existential quantifiers. It is crucial to avoid too many SMT queries and SMT queries that are too complex. Based on this observation, DuoAI introduces and combines new techniques that avoid the limitations of SMT solvers in checking invariants with existential quantifiers.

First, DuoAI runs protocol simulations at various instance sizes and logs the reached protocol states, which we call *samples*. Instance size refers to the size of distributed system (number of nodes, packets, etc.) running the protocol. These simulations are fast to execute. DuoAI directly checks candidate invariants against the samples, pruning those that do not hold to reduce the number of invariants checked by an SMT solver. To do this systematically, DuoAI introduces the *minimum implication graph*, which for a given invariant, shows all its implied weaker invariants. It then selects the *strongest* candidate invariants in the graph that hold for the samples.

Second, DuoAI combines the strongest candidate invariants with the safety property and feeds them to an SMT solver to check if the conjunction is inductive. If the check fails, it *monotonically weakens* the invariants using the graph and repeats the process until an inductive invariant is found. If the number of candidate invariants is not too large and most are required in the final solution, this method will be effective at reducing the number of SMT queries by feeding all of the candidate invariants to the SMT solver at once.

Third, DuoAI feeds the strongest candidate universal invariants, those without existential quantifiers, from the graph to an SMT solver to check if the conjunction, without the safety property, is inductive. If the check fails, it monotonically weakens the invariants using the graph, only considering candidate universal invariants, and repeats the process until the conjunction is inductive. We call this set of inductive $\forall$-only invariants the *universal core*. It then strengthens the *universal core* by iteratively adding a small subset of the strongest candidate invariants with existential quantifiers from the graph until the conjunction with the safety property is inductive. If the number of candidate invariants with existential quantifiers is large and most are not in the final solution, this method will be effective at avoiding too complex SMT queries, because it only feeds a few invariants to the SMT solver each time.

DuoAI runs these two methods for refining candidate invariants in parallel, a top-down refinement that *weakens* the candidates and a bottom-up refinement that *strengthens* the candidates, returning the result from the method that succeeds first. We prove that both methods are guaranteed to find the inductive invariant that proves the desired safety property, but they may have very different running times and resource requirements depending on the distributed protocol being verified. Using both methods together provides the best of both worlds in addressing the inefficiencies of SMT solvers.

```
1  type value
2  type quorum
3  type node
4
5  relation vote(N1:node, N2:node)
6  relation voted(N:node)
7  relation leader(N:node)
8  relation decided(N:node, V:value)
9  relation member(N:node, Q:quorum)
10 axiom forall
       Q1, Q2. exists N. member(N, Q1) & member(N, Q2)
11
12 after init {
13     voted(N) := false;
14     vote(N1, N2) := false;
15     leader(N) := false;
16     decided(N, V) := false;
17 }
18
19 action cast_vote(n1: node, n2: node) = {
20     require ~voted(n1);
21     vote(n1, n2) := true;
22     voted(n1) := true;
23 }
24
25 action become_leader(n: node, q: quorum ) = {
26     require forall N. member(N, q) -> vote(N, n);
27     leader(n) := true;
28 }
29
30 action decide(n:node, v: value) = {
31     require leader(n);
32     require forall V. ~decided(n, V);
33     decided(n, v) := true;
34 }
35
36 invariant decided(N1,V1) & decided(N2,V2) -> V1=V2
```

Figure 1: The simplified consensus protocol written in Ivy. Capitalized variables are implicitly quantified. For example, Line 16 means $\forall N:node, V:value.\ decided(N,V) := false$. "~" stands for negation.

We evaluated DuoAI using 27 widely-used distributed protocols in a head-to-head comparison against other approaches, including I4, DistAI, FOL-IC3, and SWISS. DuoAI outperforms all of the other approaches in terms of both the number of protocols for which it finds an inductive invariant and the speed at which it does so. DuoAI solves Paxos more than two orders of magnitude faster than any other approach, and is the only system that can solve more complex versions of Paxos including multi-Paxos, stoppable Paxos, and fast Paxos.

2 Overview

We use a simplified consensus protocol as an example to show how DuoAI works. Figure 1 shows the protocol written in Ivy [28], a language and tool for specifying, modeling, and verifying distributed protocols built on top of the Z3 SMT solver. Each node can vote for another node to be the leader, and when a node receives votes from a quorum of nodes, it can become the leader and decide on a value. The protocol state at any moment is represented by five relations (Lines 5-9). $vote(n_1,n_2)$

indicates whether node n_1 has voted for node n_2. $voted(n)$ indicates whether node n has ever casted a vote. $leader(n)$ indicates if n is the leader among nodes. $decided(n,v)$ indicates whether node n has decided on value v. $member(n,q)$ indicates if node n belongs to quorum q, where each quorum is a set of nodes. The axiom (Line 10) dictates a property of the member relation: any two quorums of nodes must have at least one node in common. After initialization (Lines 13-16), the protocol can non-deterministically transition from one state to another as described by the three actions *cast_vote*, *become_leader*, and *decide* (Lines 19-34). For example, $cast\_vote(n_1,n_2)$ lets a node n_1 vote for another node n_2, under the precondition that n_1 has not voted before (Line 20). Then the protocol will transition to a new state where $vote(n_1,n_2) = true$ and $voted(n_1) = true$. Finally, the safety property (Line 36) encodes the desired property of correctness of the protocol that the system cannot decide on two different values.

The safety property is an invariant of the protocol, but is not inductive as taking an action from a state satisfying the safety property may result in a new state that breaks the safety property. To verify the protocol, we need four additional invariants:

$$\forall N_1,N_2 : node.\ vote(N_1,N_2) \rightarrow voted(N_1) \tag{1}$$

$$\forall N_1,N_2,N_3 : node.\ vote(N_1,N_2) \wedge vote(N_1,N_3) \rightarrow N_2 = N_3 \tag{2}$$

$$\exists Q : quorum.\ \forall N_1,N_2 : node.\ leader(N_1) \wedge member(N_2,Q) \rightarrow vote(N_2,N_1) \tag{3}$$

$$\forall N : node.\ V : value.\ decided(N,V) \rightarrow leader(N). \tag{4}$$

The first invariant says that if a node has voted for another node, then it must be recorded as *voted* in the protocol. The second says that one node cannot vote for two different nodes. The third says that a leader must be endorsed by a quorum of nodes. More specifically, we can find a quorum Q that every node N_2 in the quorum must have voted for the leader N_1. The fourth says that only a leader can decide on a value. The conjunction of the four invariants and the safety property is inductive.

To find this inductive invariant, DuoAI simulates the protocol using different instance sizes and logs the samples. It then builds a minimum implication graph, a small fragment of which is shown in Figure 2. The full graph for simplified consensus has over 35K nodes and 170K edges. Nodes represent formulas and edges represent implication between formulas. A stronger formula will have a directed edge to an implied weaker formula. DuoAI enumerates possible candidate invariants following the graph and adds it to the candidate invariant set if it holds on the samples. For example, DuoAI checks the root node in Figure 2 and it does not hold on the samples. DuoAI then checks its implied weaker formulas, the two nodes in the second layer, iteratively going down the graph. For the simplified consensus protocol, enumeration ends with 19 candidate invariants, including equivalent forms of Eq. (1), (2), (3), and (4).

After enumeration, DuoAI runs top-down and bottom-up refinement in parallel. Top-down refinement feeds all candidate invariants and the safety property to Ivy to see if their conjunction is inductive. For simplified consensus, the conjunction is inductive, so no further weakening is required. Bottom-up refinement feeds all ∀-only invariants from the initial candidate set to Ivy then weakens them until the set of invariants is itself inductive, but may not imply the safety property. For simplified consensus, this universal core includes three invariants Eq. (1), (2), and (4). DuoAI then tries to search a small number of ∃-included invariants to add to the universal core along with the safety property so that the resulting set is inductive. DuoAI uses counterexamples from Ivy to guide the search for additional invariants and eventually identifies invariant (3) for the simplified consensus protocol, forming an inductive invariant set. For simplified consensus, top-down refinement succeeds more quickly than bottom-up refinement.

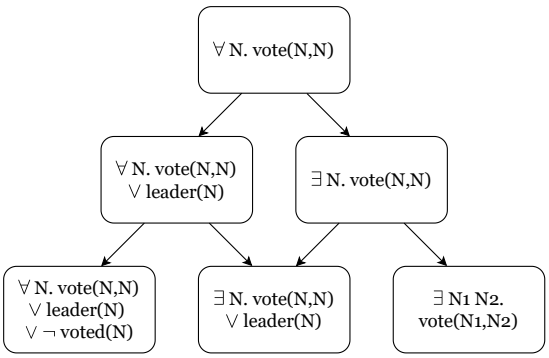

Figure 2: Fragment of the minimum implication graph for the simplified consensus protocol.

3 Minimum Implication Graph

The backbone of DuoAI is the minimum implication graph, which encodes implication relations among formulas. The graph is used to determine the order of formulas to be enumerated, and how invariants are weakened. We present formulas in prenex normal form, where the quantified variables, called the prefix, appear at the beginning of the formula followed by quantifier-free relations, called the matrix. The matrix is required to be in disjunctive normal form (DNF). For simplicity, here we only consider predicate symbols with equality. The methods can be extended to uninterpreted functions in the same manner as DistAI [38].

A formula P is strictly stronger than Q if $P \Rightarrow Q$ and $Q \not\Rightarrow P$. For two formulas $P,Q \in \mathcal{S}$, where $\mathcal{S}$ is a finite formula search space, there is a directed edge from P to Q in the minimum implication graph if and only if P is strictly stronger than Q and there is no formula R which is strictly weaker than P while strictly stronger than Q. For example, the fragment of the minimum implication graph in Figure 2 includes three formulas:

$$\forall N.\ vote(N,N) \tag{5}$$

$$\exists N.\ vote(N,N) \tag{6}$$

$$\exists N_1,N_2.\ vote(N_1,N_2) \tag{7}$$

Eq. (5) $\Rightarrow$ (6) since if $vote(N,N)$ is true for all N, there must exist some N for which it is true. Eq. (6) $\Rightarrow$ (7) since if $vote(N,N)$ is true for some N, there must exist some $N_1 = N_2$ for which $vote(N_1,N_2)$ is true. Because Eq. (5) $\Rightarrow$ (6) $\Rightarrow$ (7), there is an edge from Eq. (5) to (6), an edge from Eq. (6) to (7), but no edge from Eq. (5) to (7), because Eq. (6) is between them.

DuoAI defines the search space $\mathcal{S}$ as all formulas in disjunctive normal form for a given set of quantified variables and formula size. The formula size is defined by four parameters: *max_exists* sets the maximum number of existentially quantified variables, *max_literal* sets the upper bound of the total number of literals in the formula, *max_and* sets the maximum number of literals connected by AND, and *max_or* sets the maximum number of conjunctions connected by OR.

The minimum implication graph has two important properties as stated in Lemmas 1 and 2:

Lemma 1. *The minimum implication graph is a directed acyclic graph (DAG).*

Proof. Suppose there is a cycle $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_k \rightarrow P_1$. The edges $P_1 \rightarrow P_2, \ldots, P_{k-1} \rightarrow P_k$ imply that $P_1 \Rightarrow P_2, \ldots, P_{k-1} \Rightarrow P_k$. From the transitivity of $\Rightarrow$ we know $P_1 \Rightarrow P_k$. Since there is an edge from P_k to P_1, we know $P_1 \not\Rightarrow P_k$, a contradiction. □

Lemma 2. *For any $P,Q \in \mathcal{S}$, there is a path from P to Q in the minimum implication graph if and only if $P \Rightarrow Q \wedge Q \not\Rightarrow P$.*

Proof. We first prove the "if" direction by induction on the number of formulas in S that are strictly weaker than P while strictly stronger than Q. For the base case, if there are zero such formulas, then by definition there is an edge from P to Q. Next we prove the induction step. Suppose for any $P,Q \in S$, if $P \Rightarrow Q \wedge Q \not\Rightarrow P$, and there is no more than n formulas that are strictly weaker than P while strictly stronger than Q, then there is a path from P to Q. Now consider the case that there are $n+1$ formulas that are strictly weaker than P while strictly stronger than Q. Let R be one of the $n+1$ formulas. We know $P \Rightarrow R \wedge R \not\Rightarrow P$, and there can be no more than n formulas that are strictly weaker than P while strictly stronger than R. By the induction hypothesis, there is a path from P to R. In the same manner, we can show there is a path from R to Q. Then we concatenate the two paths and get a path from P to Q.

Next we prove the "only if" direction. If there is a path from P to Q, Let $P, F_1, F_2, ..., F_k, Q$ be the path. We know $P \Rightarrow F_1, \ldots, F_k \Rightarrow Q$, so $P \Rightarrow Q$. We prove $Q \not\Rightarrow P$ by contradiction. Suppose $Q \Rightarrow P$, then $P \Leftrightarrow Q$, so there must be an edge from F_k to P. This forms a cycle $P,F_1,...,F_k,P$, a contradiction to Lemma 1. □

To build the minimum implication graph, we need to determine the "root" nodes in the graph, that is, formulas with no predecessors since they cannot be implied by any other formula, and how to find their successors. In DuoAI, a formula $P \in S$ is added to the set of root nodes if it falls into one of two cases:

1. P has no $\exists$-quantified variable and no logical OR. For example:
$$\forall N : node.\ vote(N,N) \wedge leader(N). \tag{8}$$
2. P has unique $\exists$-quantified variables and no logical OR. For example:
$$\exists N_1, N_2 : node.\ N_1 \neq N_2 \wedge vote(N_1,N_2). \tag{9}$$

Intuitively, if a formula has an $\exists$, then by changing it to a $\forall$, we can get a stronger formula. If a formula has a logical OR, then by removing the OR and any literals followed by it, we can get a stronger formula. So in general, a root formula should have no $\exists$ and no OR, such as Eq. (8). There is one exception, represented by Eq. (9). At first sight Eq. (9) has a predecessor $\forall N_1, N_2 : node.\ N_1 \neq N_2 \wedge vote(N_1,N_2)$. However, this formula is a contradiction because $\forall N_1, N_2 : node.\ N_1 \neq N_2$ cannot be true. The minimum implication graph does not include tautologies and contradictions, so Eq. (9) itself is a root formula.

Starting from the root nodes, DuoAI incrementally builds the minimum implication graph. For formulas $P,Q \in S$, DuoAI adds an edge from P to Q if the shapes of P and Q fall into one of five cases:

1. P and Q share the same matrix. Q replaces the $\forall$-quantified variables of one type with $\exists$-quantified variables. For example:
$$P = \forall N : node, V : value.\ \neg decided(N,V)$$
$$Q = \exists N : node.\ \forall V : value.\ \neg decided(N,V).$$
2. P and Q share the same prefix. Q has one less ANDed literal than P. For example:
$$P = \text{Eq.}(8) \qquad Q = \text{Eq.}(5).$$
3. P and Q share the same prefix. Q has one more ORed conjunction than P. For example:
$$P = \forall N : node.\ vote(N,N)$$
$$Q = \forall N : node.\ vote(N,N) \vee (voted(N) \wedge leader(N)).$$
DuoAI requires that the ORed conjunction be maximal, which means it contains the maximum number of literals for the search space. The conjunction $voted(N) \wedge leader(N)$ in Q is maximal if $max\_and = 2$ or $max\_literal = 3$. For example, $Q' = \forall N : node.\ vote(N,N) \vee voted(N)$ also adds one more ORed conjunction from P, but DuoAI does not add an edge from P to Q', because Q is strictly stronger than Q'.
4. Starting from P, Q projects two $\forall$-quantified variables of the same type into one variable. For example:
$$P = \forall N_1, N_2 : node.\ vote(N_1,N_2) \vee leader(N_1)$$
$$Q = \forall N : node.\ vote(N,N) \vee leader(N).$$
5. Starting from Q, P projects two $\exists$-quantified variables of the same type into one variable. For example:
$$P = \text{Eq.}(6) \qquad Q = \text{Eq.}(7).$$

The graph constructed in this way may differ slightly from the exact minimum implication graph due to equivalent formulas. For example, formulas $\forall X.\ p(X) \vee (\neg p(X) \wedge q(X))$ and $\forall X.\ p(X) \vee q(X)$ fall into the second case, so there is an edge in the constructed graph. However, the two formulas are equivalent so there is no edge in the exact graph. We call the graph constructed by DuoAI an *approximate minimum implication graph*, whose properties are formalized in Lemmas 3, 4, and 5:

Lemma 3. *The approximate minimum implication graph is a directed acyclic graph (DAG).*

Proof. For all of the five cases, we can show that for formulas along any path in the approximate minimum implication graph, there exists one function that is strictly increasing, so there can be no cycle. For example, this is true for the function $(\#\exists\text{-}variables) - (\#\forall\ variables) + (max\_and * (\#\vee)) - (\#\wedge)$, where # denotes "the number of" (e.g., $(\#\vee)$ is the number of logical OR in a formula). □

Lemma 4. *For any $P,Q \in \mathcal{S}$, there is a path from P to Q in the approximate minimum implication graph only if $P \Rightarrow Q$.*

Proof. From the transitivity of $\Rightarrow$, we only need to show that if there is an edge from P to Q in the approximate minimum implication graph, then $P \Rightarrow Q$. This can be proved by showing $P \Rightarrow Q$ holds in each of the five cases. The first three cases are trivial. For the fourth case, in general $P = \ldots \forall X_1 X_2 \ldots matrix(X_1, X_2)$ and $Q = \ldots \forall X_1 \ldots matrix(X_1, X_1)$. Let $P' = \ldots \forall X_1 X_2 \ldots X_1 = X_2 \rightarrow matrix(X_1, X_2)$, then $P \Rightarrow P' \Leftrightarrow Q$. Similarly, for the fifth case, $P = \ldots \exists X_1 \ldots matrix(X_1, X_1)$ and $Q = \ldots \exists X_1 X_2 \ldots matrix(X_1, X_2)$. Let $Q' = \ldots \exists X_1 X_2 \ldots X_1 = X_2 \wedge matrix(X_1, X_2)$, then $P \Leftrightarrow Q' \Rightarrow Q$. □

Lemma 5. *For any formula $P \in S$ that is not a tautology or a contradiction, there exists a directed path from a root node $Q \in S$ to P in the approximate minimum implication graph.*

Proof. We prove this by construction. For a $\exists$-free formula P, if it includes no logical OR, then it is a root formula itself. Otherwise, we find the root formula Q by removing all but one ORed conjunctions. Starting from Q, we can iteratively apply the second and third cases to add conjunctions and remove literals until we reach P. For a $\exists$-included formula P, if it includes unique $\exists$-quantified variables then it is a root formula itself. Otherwise, we iteratively find a predecessor by replacing $\exists$ with $\forall$ for quantified variables of each type, until the formula becomes the $\exists$-free P'. The first case guarantees that there is a path from P' to P, and we have already shown for the $\exists$-free P', there exists a path from a root node Q. Putting it together, we have a path from Q to P in the approximate minimum implication graph. □

In other words, the approximate minimum implication graph is as useful and complete as the exact graph. DuoAI uses the approximate minimum implication graph, which, for simplicity, we will continue to refer to as the minimum implication graph unless otherwise specified.

DuoAI requires that formulas in $\mathcal{S}$ must be in a decidable fragment of first-order logic. In general, satisfiability in first-order logic is undecidable [23], so an SMT solver can get stuck in infinite instantiations and never give the *sat*/*unsat* answer. DuoAI ensures that the formulas are decidable by enforcing a fixed order of types if there is quantifier alternation (i.e., alternating $\forall$ and $\exists$) [1]. If type A is ordered before type B, then for any formula, if there exists a quantified variable V of type A, any quantified variable of type B can only occur after V if there is quantifier alternation. For example, if type *node* is ordered before type *packet*, then $\forall N : node.\ \exists P : packet$ and $\exists N : node.\ \forall P : packet$ are allowed while $\forall P : packet.\ \exists N : node$ and $\exists P : packet.\ \forall N : node$ are not. DuoAI tries to infer the order of types from the protocol specification and obtains input from the user when necessary. For example, for the simplified consensus protocol, DuoAI can infer from Line 10 that type `quorum` must be ordered before type `node`, then ask the user to place type `value` in the order. Absent user input, DuoAI will try different possible orders in parallel.

4 Candidate Invariant Enumeration

Similar to DistAI [38], DuoAI first repeatedly simulates the distributed protocol using various instance sizes, and records the reached states as samples. For example, DuoAI simulates the simplified consensus protocol on concrete instances of different numbers of values, quorums, and nodes. The simulations of different instance sizes are done in parallel and yield samples of different lengths. DuoAI follows the minimum implication graph to enumerate candidate invariants, but rather than feeding all of them to an inefficient SMT solver, it checks them directly on the samples first. A correct invariant must hold on every reachable protocol state and thus on every sample. A key difference between DuoAI and DistAI is that DuoAI keeps the original variable-length samples and uses them in invariant enumeration, while DistAI projects all samples to fixed-length vectors that it calls subsamples. The problem is that DistAI is not exhaustive in its subsampling, so that a formula with existential quantifiers that holds for DistAI's subsamples may not actually hold for the original samples. DuoAI avoids this problem by effectively considering all possible subsamples that can be derived from the original samples.

Algorithm 1 shows the enumeration algorithm, in which *pending* is a queue whose elements are formulas that will be checked on the samples, *candidates* is the set of formulas that hold on all the samples and *invalidated* is the set of formulas that do not hold on at least one of the samples. Both *candidates* and *invalidated* are initially empty (Lines 2-3), and *pending* initially consists of the root nodes of the minimum implication graph, that is, formulas that cannot be implied by any other formula. In each iteration, a formula f is popped from the *pending* queue (Line 5). If one of f's ancestors in the graph has already been added to *candidates*, DuoAI will not check f on the samples or add f to the *candidates* invariants (Lines 6-7). Oth-

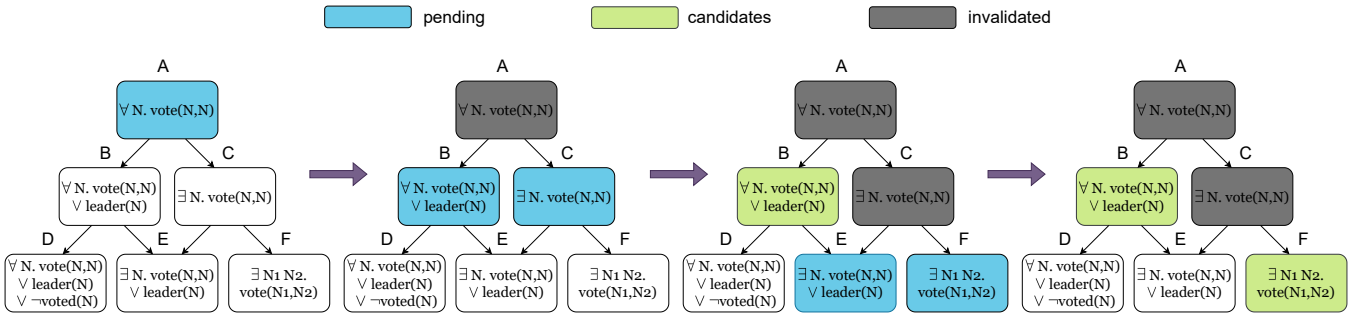

Figure 3: Invariant enumeration procedure based on the minimum implication graph. Suppose formula A and formula C do not hold on all the samples, while the other four formulas hold. Step 1: Only the root node A is in the *pending* queue. Step 2: The root node A is invalidated by the samples. We add its two successors B and C to the *pending* queue. Step 3: Formula B holds on the samples thus being added to *candidates*, while formula C is invalidated and its two successors E and F are added to the *pending* queue. Step 4: Formula E has an ancestor B which is already in *candidates*, so E is simply skipped instead of being checked on the samples. Formula F holds on the samples and is added to *candidates*.

Algorithm 1 Invariant Enumeration Algorithm

Input: Distributed protocol $\mathcal{P}$, invariant search space $\mathcal{S}$, a set of samples from protocol simulation *samples*
Output: Candidate invariants

```
1:  graph := build_minimum_implication_graph(P,S)
2:  candidates, invalidated := ∅,∅
3:  pending := graph.rootNodes
4:  while pending.notEmpty() do
5:      f := pending.dequeue()
6:      if graph.ancestors(f) ∩ candidates ≠ ∅ then
7:          continue
8:      if check_inv_holds(f, samples) then
9:          candidates := candidates ∪ {f}
10:     else
11:         invalidated := invalidated ∪ {f}
12:         for next_f ∈ graph.successors(f) do
13:             if next_f ∉ candidates and next_f ∉ invalidated
                and next_f ∉ pending then
14:                 pending.enqueue(next_f)
15: return candidates
```

erwise, DuoAI will check f on the samples and if it holds, add it to *candidates* (Lines 8-9). If f does not hold on at least one sample, DuoAI will add it to *invalidated* (Line 11), and add its successors, which are formulas weaker than f, to the *pending* queue if they have not already been added (Lines 12-14).

Figure 3 shows an example of invariant enumeration using the graph in Figure 2. DuoAI starts from the root nodes, iteratively goes down the minimum implication graph, and checks formulas against the samples. Because of this design, formulas D and E are never checked against the samples and are not added to the candidates, because their predecessor B, a formula stronger than both D and E, is already a candidate invariant. This design not only saves time checking formulas on samples, but also avoids burdening the SMT solver later with checking the inductiveness of redundant invariants. More importantly, this procedure guarantees that the resulting invariants, formulas B and F in this example, are the strongest candidate invariants that hold on the samples, which is formally stated in the following theorem:

Theorem 1. *For any correct invariant $I \in \mathcal{S}$ held by the protocol $\mathcal{P}$, at the end of invariant enumeration, either 1) $I \in candidates$, or 2) one of I's ancestors $I_{anc} \in candidates$.*

Proof. Consider three cases: 1) I has been checked on the samples, 2) I has been added to the *pending* queue but was not checked on samples, and 3) I has been never added to the *pending* queue. In the first case, since I is a correct invariant held by the protocol, it must hold on all the samples and will be added to *candidates* (Lines 8-9), so $I \in candidates$. In the second case, after I is popped from the *pending* queue, there must be an ancestor I_{anc} of I already in *candidates* (Line 6), otherwise I will be checked on the samples, so $I_{anc} \in candidates$. In the third case, we show that an ancestor $I_{anc} \in candidates$ exists. From Lemma 5, there must be a path from a root node I_0 to I, namely $I_0,I_1,...,I$. On Line 3 the root node I_0 is added to the *pending* queue. Since I_0 is added to the *pending* queue and I is not, let I_k be the last formula on the path $I_0,I_1,...,I$ that is ever added to the *pending* queue. After I_k is dequeued, there are three possible branches to take: Lines 6-7, Lines 8-9, or Lines 10-14. If it takes Lines 6-7, then there is an ancestor I_{anc} of I_k such that $I_{anc} \in candidates$. If it takes Lines 8-9, then I_k will be added to *candidates* so I_k can be the ancestor I_{anc} of I such that $I_{anc} \in candidates$. If it takes Lines 10-14, its successors will be added to the *pending* queue unless the branch condition at Line 13 evaluates to false. From our hypothesis, I_k is last formula on path $I_0,...,I_k,I_{k+1},...,I$ that is ever added to the *pending* queue. Thus, the branch condition for I_{k+1} must evaluate to false, so either $I_{k+1} \in candidates$ or $I_{k+1} \in invalidated$. However, I_{k+1} must be added to the *pending* queue before it can be added to either *candidates* or *invalidated*, a contradiction. □

Theorem 1 says that any correct invariant has either itself or

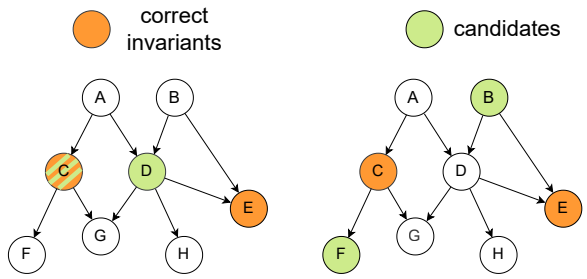

Figure 4: Possible (left) and impossible (right) candidate invariants after enumeration. Formulas C and E are correct invariants held by the protocol. It is possible that after enumeration, $candidates = \{C, D\}$ (left). Correct invariant C is in the candidate invariants itself. For correct invariant E, its ancestor D is in the candidate invariants. Theorem 1 guarantees that $candidates = \{B, F\}$ is not a possibility after enumeration, because for correct invariant C, neither itself nor its lone ancestor A is in the candidate invariants.

its ancestor (a stronger formula) in the candidate invariants. Figure 4 gives an illustration. A direct corollary is that, the set of candidate invariants after the enumeration is at least as strong as the correct invariant in $\mathcal{S}$.

5 Top-down Invariant Refinement

Based on Theorem 1, the candidate invariants can only be *too strong*, so DuoAI can monotonically weaken the candidate invariants until a correct inductive invariant is reached, which we refer to as top-down invariant refinement. Algorithm 2 shows the top-down refinement algorithm. In each iteration, DuoAI feeds the current candidate invariants to Ivy. Ivy invokes the Z3 SMT solver to check the inductiveness of each candidate invariant and the safety property. Ivy will return which invariants fail the check; if there are none, the correct inductive invariant has been found (Lines 4-5). If the safety property fails, there is no point to weaken it, and the system returns NotProvable (Lines 6-7). If one of the candidate invariants fails, DuoAI moves it from *candidates* to *invalidated* (Lines 9-10), then adds its successors (i.e., formulas that can be implied by the failed invariant) to *candidates* so long as the successor does not have a reachable ancestor in *candidates* and has not already been invalidated (Lines 12-13). An ancestor of a node is reachable if there is a path from the ancestor to the node along which no node is invalidated.

Figure 5 shows an example of top-down refinement. Suppose the current candidate invariants include formulas B and F, and by invoking the Z3 SMT solver, Ivy indicates that B is not inductive. Formulas D and E are not in *candidates*, because they can be implied by formula B which is already in *candidates*. After B is invalidated, both D and E will be added to *candidates* to let Ivy decide their inductiveness in future iterations. Alternatively, if formula F is invalidated by Ivy, no formula will be added to *candidates* because F has no successor in the minimum implication graph of search space $\mathcal{S}$.

Algorithm 2 Top-down Invariant Refinement Algorithm

Input: Distributed protocol $\mathcal{P}$, minimum implication graph *graph*, candidate invariants from enumeration CI
Output: Either an inductive invariant II, or NotProvable

```
1:  candidates, invalidated := CI, ∅
2:  while candidates.notEmpty() do
3:      failed_inv := Ivy_check(P, candidates)
4:      if failed_inv is None then
5:          return candidates
6:      else if failed_inv = safety_property then
7:          return NotProvable
8:      else
9:          candidates := candidates \ {failed_inv}
10:         invalidated := invalidated ∪ {failed_inv}
11:         for next_inv ∈ graph.successors(failed_inv) do
12:             if graph.reachable_ancestors(next_inv) ∩
                candidates = ∅ and next_inv ∉ invalidated then
13:                 candidates := candidates ∪ {next_inv}
14: return NotProvable
```

By weakening failed invariants based on the minimum implication graph rather than discarding them, DuoAI can guarantee that it never overweakens invariants to bypass the correct invariants in between. In other words, top-down refinement has a theoretical guarantee to eventually find an inductive invariant if one exists in the search space, as stated in the following theorem:

Theorem 2. *For any protocol $\mathcal{P}$ and finite search space $\mathcal{S}$, if there exists an inductive invariant $II^* \subset \mathcal{S}$ that can prove the safety property, then Algorithm 1 followed by Algorithm 2 will output such an inductive invariant II in finite time.*

Proof. The key is to prove that the while loop (Lines 2-13) maintains the following loop invariant: For any invariant $I \in II^*$, either 1) $I \in candidates$, or 2) there exists a reachable ancestor I_{anc} of I such that $I_{anc} \in candidates$. The loop invariant says that after any rounds of invariant weakening, the candidate invariants must be still at least as strong as the correct invariants. If Algorithm 2 terminates, it is impossible to have the safety property fail (Line 7). The only possibility is that a correct inductive invariant is returned (Line 5).

Theorem 1 guarantees that the loop invariant holds before entering the loop. We only need to prove that if this loop invariant holds at the beginning of round k of invariant weakening, it must still hold at the beginning of round $k+1$. This proof is done by construction for each $I \in II^*$. From the induction hypothesis, at the beginning of round k, either 1) $I \in candidates$, or 2) a reachable ancestor $I_{anc} \in candidates$. In the first case, I cannot have been invalidated during round k because $I \in II^*$, so $I \in candidates$ still holds at the beginning of iteration $k+1$. In the second case, the invalidated invariant must either be on or

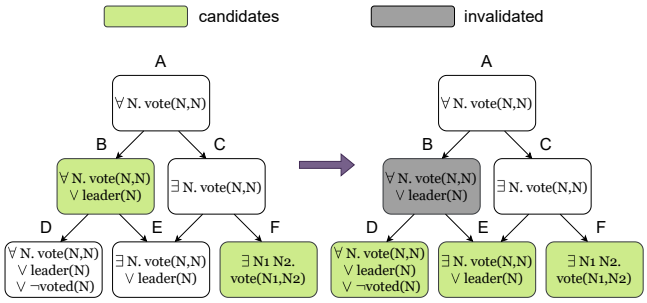

Figure 5: One round of top-down refinement. Suppose candidate invariant B fails the Ivy check. DuoAI removes B from *candidates* and adds its successors D and E to *candidates*.

not on the path from I_{anc} to I. If it is not on the path, I_{anc} remains a reachable ancestor of I and $I_{anc} \in candidates$ still holds at the beginning of iteration $k+1$. If it is on the path, let I_d be the successor of the invalidated invariant on the path. From Lines 11-13, either I_d is added to *candidates*, in which case I_d can be the new I_{anc} for iteration $k+1$, or I_d has a reachable ancestor $I_e \in candidates$, in which case we choose I_e as the new I_{anc} for iteration $k+1$. In all cases, we can find either I or a reachable ancestor I_{anc} in the candidate set, therefore the loop invariant holds.

Now we only need to prove that Algorithm 2 terminates, which follows from three observations: 1) In each loop iteration a formula is removed from *candidates* (Line 9); 2) each formula can only be added to *candidates* once (Lines 10 & 12); and 3) the formula search space $\mathcal{S}$ is finite. □

6 Bottom-up Invariant Refinement

Although top-down refinement provides a strong theoretical guarantee of finding an inductive invariant, it may take too long or run out of memory given limited computing resources if there are too many unnecessary invariants to consider. For the simplified consensus protocol in Figure 1, besides the four invariants (1)(2)(3)(4), many other invariants hold for the protocol but are unnecessary to prove the inductiveness of the safety property, for example,

$$\forall V : value, Q : quorum.\ \exists N : node.\ member(N,Q) \wedge (leader(N) \vee \neg decided(N,V)). \quad (10)$$

Invariants such as Eq. (10) do not affect the soundness of DuoAI, but they will significantly slow down the validation of candidate invariants by the SMT solver. If there are m candidate invariants, validating each invariant takes $O(m)$ time in the worst case, since the inductiveness of one invariant can depend on any other invariant, so checking all candidate invariants can take $O(m^2)$ time. Adding unnecessary invariants can increase validation time quadratically.

The key issue though is not just how many unnecessary invariants there are, but whether they have quantifier alternation (i.e., alternating ∀ and ∃), which we observe causes

Algorithm 3 Bottom-up Invariant Refinement Algorithm

Input: Distributed protocol $\mathcal{P}$, minimum implication graph *graph*, candidate invariants from enumeration *CI*
Output: Either an inductive invariant *II*, or NotProvable

Procedure 1
1: $CI_\forall := \{I | I \in CI \wedge I \text{ is } \exists\text{-free}\}$
2: $core$:= Algorithm2($\mathcal{P}$, $graph_\forall$, $CI_\forall$)
3: $noncore := CI \setminus core$

Procedure 2
4: $CE := \emptyset$
5: **for** sub **in** $powerset(noncore)$ **do**
6: **if** $\exists s \in CE$. invs_hold_on_state(sub,s) **then**
7: **continue**
8: result := Algorithm2($\mathcal{P}$, $graph$, $core \cup sub$)
9: **if** result = NotProvable **then**
10: $s \xrightarrow{a} s'$:= get_counterexample()
11: $CE := CE \cup \{s\}$
12: **else**
13: **return** result
14: **return** NotProvable

SMT solvers to struggle. For the Paxos protocol, a correct inductive invariant set of size 14 can be validated in less than a second. If we add 10 correct but unnecessary invariants with quantifier alternation, the validation will take 5 minutes. If we add 20 such invariants, the validation will take over 3 hours.In contrast, the chord ring maintenance protocol [21] with 149 ∀-only invariants only takes 8 seconds to validate.

However, a correct distributed protocol typically has a clear and human-understandable intuition, which leads to concise invariants [10]. This motivates our bottom-up invariant refinement algorithm shown in Algorithm 3. In essence, the algorithm tries to identify a small set of correct *and* helpful invariants that can eventually prove the safety property. §8 shows that the combination of bottom-up with top-down refinement provides fast performance for finding inductive invariants across a wide-range of protocols.

Algorithm 3 consists of two procedures. In Procedure 1, DuoAI first extracts all the ∀-only invariants from the candidate invariants (Line 1), which are guaranteed to be the strongest ∀-only invariants that hold on the samples. Then, DuoAI runs the top-down refinement algorithm (Line 2) using only the universal invariants and the universal portion of the minimum implication graph by removing all nodes representing existentially quantified formulas. The safety property is neglected in this top-down refinement. In this way, the ∀-only invariants are monotonically weakened until they become inductive, regardless of whether the safety property can be proved (it probably cannot). Recall that we call the now inductive ∀-only invariants the universal inductive core. DuoAI then puts every enumerated candidate invariant that is not in the universal inductive core

into *noncore* (Line 3). *noncore* mainly consists of formulas with existential quantifiers, but also includes ∀-only formulas that are not in the core, whose inductiveness may depend on ∃-included invariants. For example, for the simplified consensus protocol, the universal inductive core includes five candidate invariants, which are exactly the equivalent forms of Eq. (1), (2), and (4). There are 14 non-core candidate invariants, 13 of which have quantifier alternation, including Eq. (3) and (10).

Based on our observation that SMT solvers struggle with quantifier alternation, we expect *noncore* formulas will have a much higher cost of checking. Procedure 2 aims to identify a *small* subset of *noncore* to strengthen the candidate invariants, such that the conjunction of the universal inductive core and the subset (denoted as $core \cup sub$), or their weaker forms, can prove the safety property. Procedure 2 enumerates each subset *sub* of *noncore* (Line 5), and runs the monotonic weakening algorithm (Algorithm 2) on $core \cup sub$ (Line 8). If Algorithm 2 returns NotProvable (Line 9), DuoAI moves on to consider the next subset. Otherwise, Algorithm 2 outputs a correct inductive invariant (Line 13). The enumeration of subsets is conducted in increasing order of size, starting from the $\emptyset$, followed by all single formulas from *noncore*, then pairs, triples, and so on.

Whenever Algorithm 2 finds the safety property failed and reports NotProvable, Ivy returns a counterexample of inductiveness $s \xrightarrow{a} s'$ (Line 10), which means starting from a protocol state s satisfying the safety property and the candidate invariants, and taking an action a, the system reaches a new state s' where the safety property is violated.[1] If we view the samples from protocol simulation as positive samples on which the invariants must hold, then we can view these counterexample states s as negative samples which the invariants must exclude. DuoAI needs to identify and include another invariant I that does *not* hold on s, so that the counterexample $s \xrightarrow{a} s'$ can be excluded. When enumerating a subset of *noncore*, Procedure 2 first checks if the subset can exclude all counterexamples seen so far (Line 6). If there exists one counterexample state s on which all invariants in the subset hold, or in other words, the counterexample cannot be excluded, the monotonic weakening algorithm is bound to fail, because if a stronger invariant cannot exclude the counterexample, then its weaker forms cannot either. So Procedure 2 simply moves on to enumerate the next subset (Line 7).

For the simplified consensus protocol, when $sub = \emptyset$, the safety property fails and Ivy gives the counterexample $s = \{vote(n_1,n_1) = vote(n_1,n_2) = vote(n_2,n_1) = vote(n_2,n_2) = false,\ voted(n_1) = voted(n_2) = false,\ leader(n_1) = leader(n_2) = true,\ member(n_1, q) = member(n_2, q) = true,\ decided(n_2, v_1) = true,\ decided(n_1, v_1) = decided(n_1, v_2) = decided(n_2, v_2) = false\}$. Eq. (3) does not hold on s, so it can exclude this counterexample. In contrast, Eq. (10) holds on s, so the counterexample will persist even if Eq. (10) is added to the candidate set. Therefore, DuoAI will skip Eq. (10) and try Eq. (3), and run Algorithm 2 on its conjunction with the universal core, which gives a correct inductive invariant set consisting of Eq. (1)(2)(3)(4).

Although counterexamples can be used for top-down refinement, DuoAI currently does not because Ivy cannot return counterexamples in batch. When Ivy is configured to return a counterexample, it terminates once it identifies the first broken invariant. This is inefficient for top-down refinement, but for bottom-up refinement, counterexamples are only needed for the safety property, so DuoAI puts the safety property on top of other invariants and Ivy will give the desired counterexample.

Like top-down refinement, bottom-up refinement has a theoretical guarantee to eventually find an inductive invariant if one exists in the search space, as stated in the following theorem:

Theorem 3. *For any protocol $\mathcal{P}$ and finite search space $\mathcal{S}$, if there exists an inductive invariant $II^* \subset \mathcal{S}$ that can prove the safety property, then Algorithm 1 followed by Algorithm 3 will output such an inductive invariant II in finite time.*

Proof. We first prove that Algorithm 3 terminates in finite time. This directly follows from three facts: 1) $powerset(noncore)$ is a finite set so the for loop (Line 5) has a finite number of iterations; 2) In each loop iteration, there is at most one invocation of Algorithm 2 (Line 8); and 3) From Theorem 2, Algorithm 2 terminates in finite time.

To prove the soundness of Algorithm 3, we first observe that if Algorithm 3 outputs an invariant, it must be a correct inductive invariant, because the output must come from Algorithm 2, in which the output can only occur when the safety property is proved.

Now we prove that there will be an output invariant eventually. Observe that $noncore \in powerset(noncore)$ (Line 5). When $sub = noncore$, we have $CI \subset core \cup sub$, then Line 8 degenerates to Algorithm 2 in §5. From Theorem 2, we know a correct inductive invariant will be outputted. □

For both the top-down and bottom-up refinement, if NotProvable is returned, we know the protocol cannot be verified using invariants in the search space $\mathcal{S}$. DuoAI will try a larger search space by increasing either *max_literal*, *max_or*, *max_and*, or *max_exists*, or the per-domain number of quantified variables. By default, DuoAI alternates among the five in a round-robin manner. DuoAI sets the initial $max\_literal = 4$, $max\_or = max\_and = 3$, and $max\_exists = 1$ unless the safety property already involves $k \geq 2$ existentially quantified variables, in which case DuoAI sets $max\_exists = k$. DuoAI sets the initial number of quantified variables for domain T as the maximum number of variables of type T in any relation. For example, the relation `vote(N1:node,N2:node)` guarantees type `node` has at least two variables.

Because SMT solvers are much less efficient at checking invariants with existential quantifiers, and many distributed protocols are provable by ∀-only invariants [21], DuoAI runs

[1] In general, other than showing an invariant is not inductive, a counterexample may also show an invariant does not hold at the protocol initial state. But this cannot happen to the safety property, unless the protocol is wrong.

a $\forall$-only instance (i.e., $max\_exists = 0$) in parallel. The $\forall$-only instance only runs top-down refinement, as bottom-up refinement degenerates to the same top-down refinement (Line 2).

7 Optimizations Based on Mutual Implication

In using the minimum implication graph, DuoAI introduces several optimizations based on mutual implication relations among formulas. These relations further prune the search space and avoid redundant candidate invariants. DuoAI considers two kinds of mutual implication relations, 1) $P_1 \wedge P_2 \wedge ... \wedge P_k \Rightarrow Q$, and 2) $P_1 \wedge P_2 \wedge ... \wedge P_k \Leftrightarrow Q$. Although the latter is a special case of the former, DuoAI treats them differently. We refer to Q as a *conjunction implied formula* in the former and an *equivalently decomposable formula* in the latter. Since checking inductiveness has a quadratic complexity with the number of invariants, these optimizations have a significant improvement on efficiency.

Conjunction implied formulas. DuoAI identifies conjunction implied formulas to avoid redundant candidate invariants. Given a mutual implication relation $P_1 \wedge P_2 \wedge ... \wedge P_k \Rightarrow Q$, if all $P_1, P_2, ... P_k$ are already in the candidate invariants, DuoAI will mark Q as a conjunction implied formula and not add Q to the candidate invariants. Later during refinement, if one of $P_1, P_2, ... P_k$ is invalidated by Ivy, then the conjunction implied invariant Q is no longer redundant and will be added to the candidate invariants.

For example, suppose we have a disk replication protocol with the following three invariants:

$$\forall E : epoch, R : replica.\ crashed(E,R) \rightarrow \neg readable(E,R) \quad (11)$$

$$\forall E : epoch.\ \exists R : replica.\ readable(E,R) \quad (12)$$

$$\forall E : epoch.\ \exists R : replica.\ \neg crashed(E,R). \quad (13)$$

One can check that among Eq. (11)(12)(13), no formula can imply another. But the conjunction of Eq. (11) and (12) can imply Eq. (13). This is because Eq. (12) says that for every epoch E, there must be a readable replica R. Then from Eq. (11), the readable replica R cannot be crashed. Therefore, for every epoch E, there must be a replica R that does not crash, which is expressed by Eq. (13). If Eq. (11) and (12) are already candidate invariants, DuoAI will mark Eq. (13) as a conjunction implied formula and not add it to the candidate invariants.

There are many classes of mutual implication relations in first-order logic. DuoAI identifies three classes of conjunction implied formulas to prune candidate invariants; in each class, the first two formulas mutually imply the third:

1. Replace a literal with a weaker literal, as discussed in the example Eq. (11)(12)(13):

$$P_1 = \forall X.\ r(X) \rightarrow s(X)$$
$$P_2 = prefix.\ (r(X) \wedge ...) \vee ...$$
$$Q = prefix.\ (s(X) \wedge ...) \vee ...$$

2. Conjunct a literal with a weaker literal:

$$P_1 = \forall X.\ r(X) \rightarrow s(X)$$
$$P_2 = prefix.\ (r(X) \wedge ...) \vee ...$$
$$Q = prefix.\ (r(X) \wedge s(X) \wedge ...) \vee ...$$

3. "Merge" a $\forall$ formula and an $\exists$ formula:

$$P_1 = \exists X.\ r(X)$$
$$P_2 = \forall X.\ s(X) \vee ...$$
$$Q = \exists X.\ (r(X) \wedge s(X)) \vee ...$$

In all three classes, r and s can be generalized to conjunctions (e.g., $r_1(X) \wedge \neg r_2(X)$, $\neg s_1(X) \wedge s_2(X) \wedge s_3(X)$). A key advantage of this optimization is that given a finite search space, DuoAI can identify conjunction implied formulas based on invariants within that search space, even though the conjunction of invariants is not in that search space.

Equivalently decomposable formulas. DuoAI also identifies equivalently decomposable formulas to avoid redundant candidate invariants. Given a mutual implication relation $P_1 \wedge P_2 \wedge ... \wedge P_k \Leftrightarrow Q$, DuoAI will mark Q as an equivalently decomposable formula and never add Q to the candidate invariants. Later during refinement, if one of $P_1, P_2, ... P_k$ is invalidated by Ivy, Q will also be invalidated and therefore there is never any reason to consider Q further as a candidate invariant.

For example, suppose the disk replication protocol has invariant:

$$\forall E : epoch.\ \exists R_1, R_2 : replica.\ readable(E,R_1) \wedge writable(E,R_2). \quad (14)$$

There is no need to ever include such an invariant in the candidate set, because it is equivalently decomposable to invariants (15) and (16).

$$\forall E : epoch.\ \exists R : replica.\ readable(E,R) \quad (15)$$

$$\forall E : epoch.\ \exists R : replica.\ writable(E,R). \quad (16)$$

However, suppose we slightly modify invariant (14) to one of the following two formulas:

$$\forall E : epoch.\ \exists R_1 : replica.\ readable(E,R_1) \wedge writable(E,R_1) \quad (17)$$

$$\forall E : epoch.\ \exists R_1, R_2 : replica.\ R_1 \neq R_2 \wedge readable(E,R_1) \wedge writable(E,R_2). \quad (18)$$

These invariants are not equivalently decomposable to invariants (15) and (16). Take Eq. (17) as an example. One can verify that (17) $\Rightarrow$ (15) $\wedge$ (16), but (15) $\wedge$ (16) $\not\Rightarrow$ (17), because Eq. (17) requires the same replica to be both readable and writable, while for Eq. (15) and (16), it is possible to have one readable replica and a different writable replica.

DuoAI identifies if a formula is equivalently decomposable based on the structure of the formula itself by considering two classes of equivalently decomposable formulas. One class is embodied by the following lemma:

Lemma 6. *For a formula F in prenex and disjunctive normal form, we build a graph G_C for each conjunction C in F. G_C has one node for each literal, and an edge between two literals if and only if they share an existentially quantified variable. If for some C in F, the graph G_C has $k \geq 2$ connected components, then F is equivalently decomposable into k formulas.*

Proof. For simplicity, here we show the proof for $k = 2$ (i.e., the literals in C form two connected components). If there are $k > 2$ connected components, then the same analysis below will show that formula F is equivalently decomposable into k formulas.

Literals that share $\exists$-variables must be in the same connected component. So we can divide the $\exists$-variables into two sets $\{Y_1,...,Y_m\}$ and $\{Z_1,...,Z_n\}$. The first connected component can only include $\exists$-variables $Y_1,...Y_m$, while the second can only include $Z_1,...,Z_n$. Assume the quantifier prefix has shape $\forall X_1 ... X_s \exists Y_1 ... Y_m Z_1 ... Z_n$. We use $\vec{X},\vec{Y},\vec{Z}$ to denote $X_1 ... X_s$, $Y_1 ... Y_m$, and $Z_1 ... Z_n$. The proof can be generalized to any alternating $\forall/\exists$ using skolemization.

Let $f(\vec{X}, \vec{Y})$ be the disjunction of literals in the first connected component, and $g(\vec{X}, \vec{Z})$ be the disjunction of literals in the second connected component. Let $h(\vec{X},\vec{Y},\vec{Z})$ be the disjunction of all conjunctions other than C in formula F. Then formula F can be rewritten as:

$$\forall\vec{X}\exists\vec{Y}\vec{Z}.\ (f(\vec{X},\vec{Y})\wedge g(\vec{X},\vec{Z}))\vee h(\vec{X},\vec{Y},\vec{Z}). \tag{19}$$

We now show that, Eq. (19) is equivalently decomposable into:

$$\forall\vec{X}\exists\vec{Y}\vec{Z}.\ f(\vec{X},\vec{Y})\vee h(\vec{X},\vec{Y},\vec{Z}) \tag{20}$$

$$\forall\vec{X}\exists\vec{Y}\vec{Z}.\ g(\vec{X},\vec{Z})\vee h(\vec{X},\vec{Y},\vec{Z}). \tag{21}$$

It is trivial that Eq. (19) implies both Eq. (20) and (21). We now show the interesting direction — the conjunction of Eq. (20) and (21) implies Eq. (19). Suppose both Eq. (20) and (21) hold. For any $\vec{X}$, consider two cases: 1) $\exists\vec{Y}\,\vec{Z}.\ h(\vec{X},\vec{Y},\vec{Z})$. In this case Eq. (19) directly holds. 2) $\forall\vec{Y}\,\vec{Z}.\ \neg h(\vec{X},\vec{Y},\vec{Z})$. Then according to Eq. (20), $\exists\vec{Y}_1\vec{Z}_1.\ f(\vec{X},\vec{Y}_1)$. Similarly, from Eq. (21), $\exists\vec{Y}_2\vec{Z}_2.\ g(\vec{X},\vec{Z}_2)$. If we select $\vec{Y}_1$ and $\vec{Z}_2$, then we have $f(\vec{X},\vec{Y}_1)\wedge g(\vec{X},\vec{Z}_2)$, so Eq. (19) still holds. Putting the two cases together, when both Eq. (20) and (21) are true, Eq. (19) must be true. □

The proof also gives an algorithm to find the k decomposed formulas. Figure 6 shows how to apply Lemma 6 on the aforementioned formulas. For Eq. (14), the two literals $readable(E,R_1)$ and $writable(E,R_2)$ share no $\exists$-quantified variable (E is $\forall$-quantified), so there is no edge between the two literals. The graph has two connected components, so Eq. (14) is equivalently decomposable into two formulas (Eq. (15)(16)). For Eq. (17), the two literals share an $\exists$-quantified variable R_1, so there is an edge between the two literals and the graph is connected, which means Eq. (17) cannot be decomposed in this way. The same analysis can be applied to Eq. (18).

We note a corollary of Lemma 6. For an $\exists$-free formula, it is equivalently decomposable if it has any conjunction.

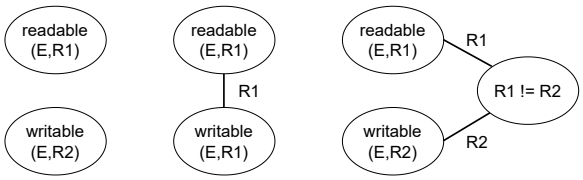

Figure 6: Checking equivalently decomposability of formulas (14)(17)(18) (from left to right).

For example, $\forall X.\ p(X) \wedge q(X)$ is equivalent with the pair $\forall X.\ p(X)$ and $\forall X.\ q(X)$. This indicates that we do not need to consider any conjunction when enumerating $\forall$-only formulas, a significant reduction in search space.

The other class of equivalently decomposable formulas that DuoAI identifies is embodied in the following:

$$\forall X_1 X_2.\ matrix(X_1,X_2) \tag{22}$$

$$\forall X_1.\ matrix(X_1,X_1) \tag{23}$$

$$\forall X_1 X_2.\ X_1 \neq X_2 \rightarrow matrix(X_1,X_2) \tag{24}$$

One can check that Eq. (22) is equivalently decomposable to Eq. (23) and (24), and will therefore not be added as a candidate invariant. In general, DuoAI only considers formulas whose leading $\forall$-quantified variables are unique. Similar optimizations have been used in DistAI [38].

8 Evaluation

Experimental setup. To demonstrate the performance of DuoAI, we implemented and evaluated DuoAI on 27 distributed protocols from multiple sources [12, 13, 21, 27, 28], including those that can only be proved by inductive invariants with $\exists$-quantifiers. The DuoAI implementation consists of 6.1K lines of C++ code for invariant enumeration and refinement, compiled by gcc 7.5.0, and 2.3K lines of Python code running with Python 3.8.10 for protocol simulation. For comparison, we also ran 6 other invariant inference tools: SWISS [10], IC3PO [8, 9], FOL-IC3 [13], DistAI [38], UPDR [12], and I4 [21]. All experiments were performed on a Dell Precision 5829 workstation with a 4.3GHz 28-core Intel Xeon W-2175, 62GB RAM, and a 512GB Intel SSD Pro 600p, running Ubuntu 18.04.

We configured the alternative invariant inference tools following their best practices. SWISS requires the user to bound the search space by specifying 4 parameters, including the number of existentially quantified variables and the number of literals in a formula. For every protocol, we use the same parameter settings as in SWISS's own evaluation [10]. IC3PO and I4 require the user to specify a finite instance size for their model checkers to work on. For IC3PO, we only specified the minimum size and the tool itself could determine how to increase the instance size. For I4, we started from the minimum size where the protocol can function and iteratively increased

the instance size upon failure (e.g., $node = 2, node = 3, ...$). FOL-IC3 provides a ∀-only mode and a default mode. We ran both and report the runtime of whichever succeeded first.

Results summary. Table 1 shows the running time in seconds for each tool on each protocol. For each protocol, we also report the number of relations and lines of code in its Ivy specification; for example, Figure 1 is a simplified version of consensus epr. The top portion of the table shows protocols provable with a ∀-only inductive invariant, while the bottom portion shows protocols that can only be proved with a ∃-included inductive invariant. We allowed each tool to spend up to an entire week trying to solve each protocol. For protocols that a tool fails to solve, we report "fail" if the tool terminated without an inductive invariant, "error" if the tool itself returned an error, "Z3 error" if the underlying SMT solver used returned an error, "memout" if the tool ran out of memory and terminated, and "timeout" if the tool did not complete within a week.

DuoAI dominates all other tools in the number of protocols it solves, solving all but 1 of the 27 protocols. SWISS cannot solve 8 protocols, FOL-IC3 and IC3PO cannot solve 9 protocols, DistAI and UPDR cannot solve 13 protocols, while I4 cannot solve 15 protocols. DuoAI is the only tool that solves all ∀-only protocols, is the only tool that solves Paxos as well as all other non-Paxos protocols with ∃ quantifiers, and is the only tool that solves 3 of the more complex Paxos variants, including multi-Paxos, stoppable Paxos, and fast Paxos. There were no protocols solvable by another tool that were not solved by DuoAI.

DuoAI also dominates all other tools in how fast it solves the protocols, solving 15 of the protocols faster than any other tool. DuoAI is faster than SWISS on all but 3 of the protocols solved by SWISS, is faster than IC3PO on all but 3 of the protocols solved by IC3PO, is faster than FOL-IC3 on all but 2 of the protocols solved by FOL-IC3, and is faster than UPDR on all of the protocols solved by UPDR. DuoAI is up to 3 orders of magnitude faster than each of these protocols. DuoAI is faster than DistAI on all but 5 of the protocols solved by DistAI, and is faster than I4 on all but 2 of the protocols solved by I4. DuoAI is up to an order of magnitude faster than either DistAI or I4. The speed differences versus DistAI and I4 appear less in part because neither could solve any of the protocols with existential quantifiers. In most cases in which DuoAI is slower than other protocols, it is by at most a few seconds.

Detailed comparison and discussion. For the protocols provable with ∀-only invariants, DuoAI is the only tool that solves all 15 protocols. On ∀-only protocols, DuoAI's ∀-only instance is similar to DistAI, without subsampling and with mutual implication optimization and parallelism in simulation. DuoAI beats DistAI on 10 protocols. Unlike DuoAI, DistAI times out on ticket lock, which we discovered is due to a bug in the implementation of its protocol simulation. Chord ring maintenance is the only protocol on which DuoAI is much slower than DistAI. DistAI only allows invariants as disjunction of literals, and implements an invariant as a `vector<int>`. In contrast, DuoAI considers invariants in disjunctive normal form, so an invariant is a disjunction of conjunction of literals, implemented as a `set<vector<int>>`. This makes invariant operation slower in DuoAI. Chord ring maintenance is the only ∀-provable protocol that takes significant time on candidate invariant enumeration so the overhead is exacerbated.

For the protocols that require invariants with ∃-quantifiers to prove, DuoAI solves 11 out of 12 protocols, more than any other tool. DuoAI only fails on vertical paxos, which other tools also fail on. DistAI, I4, and UPDR fail on all of these protocols because they can only generate ∀-only invariants. IC3PO solves 4 protocols and fails on 3 protocols, but runs out of memory on 6 protocols, because it requires model checking to infer invariants on a finite instance. For more complex protocols like fast Paxos, the model checker requires too large of an instance size. In contrast, DuoAI searches in formula space and its performance does not depend (exponentially) on instance size.

For the complex Paxos-family protocols, only SWISS also verified Paxos and flexible Paxos, though it required several hours to do so. All other tools failed on all Paxos-family protocols. In contrast, DuoAI verified Paxos and flexible Paxos in less than 2 minutes. Only DuoAI verified multi-Paxos, stoppable Paxos, and fast Paxos.

As the only other tool that solves Paxos, it is instructive to compare SWISS with DuoAI. Similar to DuoAI, SWISS also enumerates candidate invariants given a bounded search space and checks their inductiveness using the SMT solver. However, it has two fundamental differences compared with DuoAI. First, SWISS relies exclusively on the SMT solver to tell the correctness of invariants, while DuoAI also uses the samples from protocol simulation to filter out invalid invariants. As we demonstrated in §6, SMT calls can be expensive with quantifier alternation and will negatively affect performance. Second, SWISS struggles to find mutually inductive invariants, i.e., a bundle of invariants that are inductive together but none are inductive individually. This is because SWISS can only build the invariant set by adding one and only one invariant each time and keep the set inductive. In the lock server async and the sharded key-value store protocols, where mutually inductive invariants are required to prove the safety property, SWISS has to manually increase the maximum number of literals from 6 to 9. This allows the mutually inductive invariants to be conjuncted into one big invariant, but results in a much larger search space and long runtimes of 44 and 128 minutes, respectively. DuoAI enumerates candidate invariants following the minimum implication graph and generates the strongest candidate invariants. The mutually inductive invariants (or their stronger forms) are guaranteed to be in the candidate invariants together. DuoAI solved both protocols within 2 seconds.

For the vertical paxos protocol, the human-expert inductive invariants include an invariant with 8 literals. Even after the optimization based on mutual implication, it still has 7

Distributed protocol	Relations	LoC	DuoAI	SWISS	IC3PO	FOL-IC3	DistAI	UPDR	I4
chord ring maintenance	8	123	200.9	timeout	17.1	timeout[c]	58.0	Z3 error	673
consensus forall	7	55	11.9	40.3	457	1500	15.6	59.0	122
consensus wo decide	6	46	3.9	26.1	160	24.8	8.5	24.8	27.5
database chain replication	13	96	9.5	108951	4.5	559	90.3	57.6	66.6
decentralized lock	2	21	9.9	5.8	24.3	69.0	10.2	51.0	20.7
distributed lock	4	43	6.1	timeout	12856	1660	15.3	63568	195
ring leader election	3	45	3.5	14.3	memout	10.8	2.9	103	5.3
learning switch ternary	4	45	14.2	308	23.8	timeout	24.7	1334	12.9
learning switch quad	2	21	52.4	1322	63.6	timeout	372	273	memout
lock server async	5	45	1.9	2625	5.6	4.8	1.1	4.4	8.7
lock server sync	2	21	1.3	1.0	3.2	1.0	0.9	3.3	0.6
sharded key-value store	3	31	1.9	7662	5.4	9.7	1.2	3.5	error
ticket lock	5	49	23.9	fail	56.2	58.1	timeout	143	fail
toy consensus forall	4	27	1.9	5.9	3.0	5.4	3.1	3.4	9.4
two-phase commit	7	70	1.5	9.1	4.7	4.8	2.0	9.4	10.2
client server ae	4	28	1.5	5.2	2.3	355	timeout	fail	fail
client server db ae	7	48	3.1	33.7	memout[i]	4822	timeout	fail	fail
consensus epr	7	52	4.8	28.8	1118	471	timeout	fail	memout
hybrid reliable broadcast	12	120	1211.2	fail	memout[i]	931	error	fail	error
sharded kv no lost keys	3	32	2.1	1.8	4.8	3.7	timeout	fail	error
toy consensus epr	4	25	2.6	4.3	2.4	32.9	timeout	fail	fail
Paxos	9	75	60.4	16665	fail[i]	timeout	timeout	timeout	memout
flexible Paxos	10	77	78.7	28337	memout[i]	timeout	timeout	fail	memout
multi-Paxos	10	91	1549	timeout	fail	timeout	timeout	timeout	memout
stoppable Paxos	11	118	4051	error	fail	timeout	error	timeout	error
fast Paxos	12	102	26979	timeout	memout	memout	timeout	fail	error
vertical Paxos	12	120	memout	timeout	memout	memout	error	fail	error

c The SWISS authors reported that FOL-IC3 solved chord ring maintenance [10], but we found that the `chord.pyv` file they used has 3 bugs.
i The IC3PO authors [8, 9] reported that IC3PO succeeded on client server db ae (17 s), hybrid reliable broadcast (587 s), Paxos (568 s), and flexible Paxos (561 s). However, they retrofitted the protocols and manually provided clauses with quantifier alternation that could appear in the invariants, which is difficult to do without first knowing the ground-truth invariants. The 4 protocols have much simpler inductive invariants when expressed on top of these clauses, with all except the simplest, client server db ae, becoming $\exists$-free. Ivy fails when checking the invariants generated by IC3PO for Paxos and flexible Paxos. The IC3PO authors [9] imply that the invariants had to be manually checked against the human-expert invariants.

Table 1: Comparison of different tools for finding inductive invariants for 27 distributed protocols (running time in seconds).

literals. Under the minimum per-domain number of quantified variables that can encode the human-expert invariants, there are 60 predicates that can appear in the invariants. Considering their negations, the size of the invariant search space is at the magnitude of $120^7 \approx 4e14$, well exceeding the computational power of a normal workstation. In comparison, for fast paxos, the largest invariant includes 5 literals, and there are 38 predicates. The size of the search space is at the magnitude of $3e9$. For vertical Paxos, DuoAI ends with a universal core and a set of checked non-core invariants when exhausting memory. These invariants are inductive and can be utilized as hints, although they cannot imply the safety property.

As explained in §6, DuoAI runs top-down refinement, bottom-up refinement, and a $\forall$-only instance in parallel. Not surprisingly, the $\forall$-only instance generates the inductive invariants first for all 15 protocols that do not require existential quantifiers. Among the 11 protocols solved by DuoAI that require existential quantifiers, the top-down refinement gives the inductive invariants first for the 5 simpler protocols — client server ae, client server db ae, toy consensus epr, consensus epr, and sharded kv no lost keys. The bottom-up refinement also succeeded but took longer. For example, for client server db ae, there are 8 candidate invariants in *noncore*. A subset of size 3 was sufficient to prove the safety property. However, the bottom-up refinement would first enumerate and fail on all single invariants and pairs before enumerating the correct triple. This takes more than 3 times longer than top-down refinement, in which after a single round of weakening, DuoAI found an inductive invariant.

For the 6 more complicated protocols with existential quantifiers, including hybrid reliable broadcast and the 5 Paxos-family protocols solved, only the bottom-up refinement generated the inductive invariants. The top-down refinement got stuck at checking the inductiveness of the invariants. For example, for multi-Paxos, after enumeration, DuoAI has a candidate invariant set of size 615, and 581 of them have quantifier alternation. Checking inductiveness of this many formulas is a hopeless task for the Z3 SMT solver. However, to prove the safety property, only 2 of the 581 candidate invariants are needed. In the bottom-up refinement, each time only a

small subset of *noncore* invariants conjuncted with the ∀-only core are fed to Ivy, so the Z3 SMT solver could handle the candidate invariants. For Paxos, flexible Paxos, multi-Paxos, and stoppable Paxos, a subset of size 2 were sufficient, while a subset of size 6 was needed for fast Paxos. This also validates our assumption that real-world distributed protocols should have concise invariants, and should not require too many invariants with quantifier alternation to verify.

Limitations By requiring quantifier alternation to conform to a fixed order of types, DuoAI ensures that the verification condition is in a decidable fragment of first-order logic. However, without decidability, an SMT solver may still succeed. For client server db ae and hybrid reliable broadcast, the invariants written by human experts are not in a decidable fragment, yet they can be efficiently verified by the SMT solver. For both protocols, DuoAI found alternative inductive invariants within the decidable fragment. If a protocol cannot be verified in decidable logic, DuoAI will fail to prove it.

9 Related Work

Many studies [11, 17, 20, 31, 34–36] verify the correctness of distributed protocols with manually given inductive invariants. Early systems [12, 21, 29, 38] for automatically inferring inductive invariants do not work for invariants with ∃-quantifiers which are required for protocols such as Paxos, though pdH [29] can find inductive invariants for retrofitted Paxos-family protocols without ∃-quantifiers.

Recent systems consider invariants with ∃-quantifiers. FOL-IC3 [13] generates an invariant candidate that can separate a positive and negative example set, and iteratively adds more examples until the invariant is correct. It has no theoretical guarantee of success. Its heavy use of SMT queries to validate as well as synthesize invariants makes it slow in practice, timing out on even protocols with ∀-only inductive invariants.

SWISS [10] iteratively strengthens an invariant by adding small inductive formulas until the invariant is strong enough to prove the safety property. It was the first tool to automatically verify Paxos. Its inefficiency in exploring the search space and inability to infer mutually inductive invariants make it fail on several protocols solved by alternative tools.

IC3PO [8, 9] applies model checking on a finite instance similar to I4, while adding support to generalize existentially quantified invariants. The model checker functions well on small instances, but frequently exhausts memory on complex protocols that require larger instances, as shown in Table 1.

P-FOL-IC3 [14] is concurrent work that extends FOL-IC3 by exploiting parallelism in formula search, introducing the invariant-friendly *k-Term Pseudo-DNF* to bound the search space, and randomly guessing some not-yet-inductive formulas to be eventually inductive, forcing their counterexamples to be excluded. P-FOL-IC3 has no theoretical guarantee and is less robust in practice due to its randomized nature; it failed in three out of five trials on stoppable Paxos, and two out of five trials on fast Paxos.

The tools discussed above, along with DuoAI, only verify safety properties of distributed protocols. Complementary work has explored connecting verification of protocols to their practical implementations [11, 31], and verifying liveness properties of distributed protocols [26].

AutoML [5, 18, 33] searches for machine learning models and hyperparameters, which may appear similar to finding inductive invariants. However, the inductiveness of invariants has strong correlation, which is difficult to capture for AutoML.

Many automated invariant inference tools have been built for other domains, mostly on learning loop invariants to verify sequential programs [3, 4, 6, 7, 15, 24, 25, 30, 32, 37, 39]. Invariant inference has been used to prove properties on inductive algebraic data types [16, 22], integer linear dynamical systems [19], and deep neural networks [2]. None of these methods consider nondeterminism in concurrent or distributed settings, thus they cannot be directly applied to distributed protocols.

10 Conclusions

DuoAI automatically and efficiently infers inductive invariants for verifying distributed protocols by reducing SMT costs. It introduces the minimum implication graph to define the structure of the invariant search space. This enables efficient enumeration of possible invariants, which are checked on samples from protocol simulation to reduce SMT queries. DuoAI guarantees that the enumerated candidate invariants are at least as strong as any correct invariants. DuoAI then runs top-down and bottom-up refinement in parallel. The former monotonically weakens the candidate invariants following the minimum implication graph. The latter divides the candidate invariants into an SMT-friendly universal inductive core and other noncore invariants, and searches for a small subset of noncore invariants that can be added to the core to prove the safety property of the protocol. Both top-down and bottom-up refinement have strong theoretical guarantees for finding inductive invariants, and their combination is effective at reducing SMT query costs for invariants with existential quantifiers. DuoAI dominates alternative tools in both the number of protocols it verifies and the speed at which it does so, including giving automated proofs for several Paxos variants.

11 Acknowledgments

Ji-Yong Shin provided helpful comments on earlier drafts. This work was supported in part by three Amazon Research Awards, a Guggenheim Fellowship, DARPA contract N66001-21-C-4018, and NSF grants CCF-1918400, CNS-2052947, and CCF-2124080. Ronghui Gu is the Founder of and has an equity interest in CertiK.

References

[1] Decidability in Ivy. http://microsoft.github.io/ivy/decidability.html.

[2] Guy Amir, Michael Schapira, and Guy Katz. Towards scalable verification of deep reinforcement learning. In *Proceedings of the 21st Conference on Formal Methods in Computer Aided Design (FMCAD '21)*, pages 193–203, October 2021.

[3] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Solving constrained Horn clauses using syntax and data. In *Proceedings of the 18th Conference on Formal Methods in Computer Aided Design (FMCAD '18)*, pages 1–9, October 2018.

[4] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV '19)*, pages 259–277, July 2019.

[5] Matthias Feurer, Aaron Klein, Jost Eggensperger, Katharina Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Proceedings of the 29th Conference on Neural Information Processing Systems (NIPS '15)*, pages 2962–2970, December 2015.

[6] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Learning universally quantified invariants of linear data structures. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV '13)*, pages 813–829, July 2013.

[7] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, page 499–512, January 2016.

[8] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *Proceedings of the 13th NASA Formal Methods Symposium (NFM '21)*, pages 131–150, May 2021.

[9] Aman Goel and Karem A Sakallah. Towards an automatic proof of Lamport's Paxos. In *Proceedings of the 21st Conference on Formal Methods in Computer Aided Design (FMCAD '21)*, pages 112–122, October 2021.

[10] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It's a small (enough) world after all. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*, pages 115–131, April 2021.

[11] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, pages 1–17, October 2015.

[12] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM*, 64(1), March 2017.

[13] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, page 703–717, September 2020.

[14] Jason R. Koenig, Oded Padon, Sharon Shoham, and Alex Aiken. Inferring invariants with quantifier alternations: Taming the search space explosion. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '22)*, pages 338–356, April 2022.

[15] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10)*, pages 328–343, November 2010.

[16] Yurii Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich. Beyond the elementary representations of program invariants over algebraic data types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, page 451–465, June 2021.

[17] Leslie Lamport. Byzantizing Paxos by refinement. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC '11)*, pages 211–224, September 2011.

[18] Trang T Le, Weixuan Fu, and Jason H Moore. Scaling tree-based automated machine learning to biomedical big data with a feature set selector. *Bioinformatics*, 36(1):250–256, January 2020.

[19] Engel Lefaucheux, Joël Ouaknine, David Purser, and James Worrell. Porous invariants. In *Proceedings of 33rd International Conference on Computer Aided Verification (CAV '21)*, pages 172–194, July 2021.

[20] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16), pages 357–370, January 2016.

[21] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, pages 370–384, October 2019.

[22] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. Data-driven inference of representation invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, pages 1–15, June 2020.

[23] Donald Monk. *Mathematical Logic*. Springer, October 1976.

[24] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. Counterexample-guided approach to finding numerical invariants. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE '17)*, pages 605–615, August 2017.

[25] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, page 42–56, June 2016.

[26] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. Reducing liveness to safety in first-order logic. *Proceedings of the ACM on Programming Languages*, 2(POPL), January 2018.

[27] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017.

[28] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, pages 614–630, June 2016.

[29] Oded Padon, James R Wilcox, Jason R Koenig, Kenneth L McMillan, and Alex Aiken. Induction duality: Primal-dual search for invariants. *Proceedings of the ACM on Programming Languages*, 6(POPL), January 2022.

[30] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: Learning loop invariants with continuous logic networks. In *Proceedings of 8th International Conference on Learning Representations (ICLR '20)*, March 2020.

[31] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, 2(POPL), January 2018.

[32] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori. A data driven approach for algebraic loop invariants. In *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, pages 574–592, March 2013.

[33] Christian Steinruecken, Emma Smith, David Janz, James Lloyd, and Zoubin Ghahramani. The automatic statistician. In *Automated Machine Learning*, pages 161–173. Springer, May 2019.

[34] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, pages 662–677, June 2018.

[35] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, pages 357–368, June 2015.

[36] Doug Woos, James R Wilcox, Steve Anton, Zachary Tatlock, Michael D Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CCP '16)*, pages 154–165, January 2016.

[37] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, pages 106–120, June 2020.

[38] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-driven automated invariant learning for distributed protocols. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*, pages 405–421, July 2021.

[39] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*, pages 707–721, June 2018.