DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan, *Columbia University*

https://www.usenix.org/conference/osdi21/presentation/yao

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols

Jianan Yao Runzhou Tao Ronghui Gu Jason Nieh Suman Jana Gabriel Ryan
Columbia University

Abstract

Distributed systems are notoriously hard to implement correctly due to non-determinism. Finding the inductive invariant of the distributed protocol is a critical step in verifying the correctness of distributed systems, but takes a long time to do even for simple protocols. We present DistAI, a data-driven automated system for learning inductive invariants for distributed protocols. DistAI generates data by simulating the distributed protocol at different instance sizes and recording states as samples. Based on the observation that invariants are often concise in practice, DistAI starts with small invariant formulas and enumerates all strongest possible invariants that hold for all samples. It then feeds those invariants and the desired safety properties to an SMT solver to check if the conjunction of the invariants and the safety properties is inductive. Starting with small invariant formulas and strongest possible invariants avoids large SMT queries, improving SMT solver performance. Because DistAI starts with the strongest possible invariants, if the SMT solver fails, DistAI does not need to discard failed invariants, but knows to monotonically weaken them and try again with the solver, repeating the process until it eventually succeeds. We prove that DistAI is guaranteed to find the ∃-free inductive invariant that proves the desired safety properties in finite time, if one exists. Our evaluation shows that DistAI successfully verifies 13 common distributed protocols automatically and outperforms alternative methods both in the number of protocols it verifies and the speed at which it does so, in some cases by more than two orders of magnitude.

1 Introduction

Distributed systems are hard to design and implement correctly. This is due to the intrinsic non-determinism from asynchronous node communications and various failure scenarios. Formal verification techniques offer a solution by proving that a distributed system is correct under all circumstances [10, 16, 32]. The verification of distributed systems consists of two components: i) proving that the desired safety properties hold for the distributed protocol itself, and ii) proving that the protocol implementation is correct.

While much work has focused on proving a system correctly implements a protocol [10, 16, 31–33], we focus on proving the protocol itself has the desired safety properties. 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 [10] is that no two nodes in the system hold a lock at the same time. The typical proof strategy is to prove that an invariant that implies the safety property is inductive, meaning that if the system starts from a state that satisfies the invariant, the invariant will still hold for any state that is reachable via a valid transition from the previous state. If the safety property itself is inductive, the proof is done. However, this is not true for almost all nontrivial distributed protocols, so that the proof requires finding an invariant that implies the safety property, then proving that it is inductive.

Finding the inductive invariant for distributed protocols is difficult, taking a long time for even simple protocols [18]. IVy [24] provides an interactive tool to make this easier. A developer provides a set of invariants and protocol specification that defines its safety property, which IVy automatically checks using an SMT solver. Each invariant can be expressed as a logical formula, which consists of a prefix with quantifiers (∀ or ∃) and a certain number of variables, and a set of logical relations among the variables. IVy checks if adding the invariants to the safety property makes it inductive, meaning that the conjunction of all invariants with the safety property is inductive. Conjunction requires each invariant to hold, so IVy reports whether any invariant fails, at which point the developer can try again with a different set of invariants. This requires substantial manual effort by the developer.

Recent work has focused on automating invariant generation for distributed protocols, but with various limitations. I4 [18] observes that invariants for some distributed protocols do not depend on the size of the system, so I4 uses a specialized model checker to generate invariants for a small size system, then generalizes them and uses IVy to check if the conjunction of the invariants with the safety property is inductive for the protocol specification. If not, IVy indicates which invariants failed. I4 removes them and tries again, and if that fails, tries using a larger size system to generate invariants. However, I4 provides no guarantee that it can find the inductive invariant, as it may not be possible to verify a protocol based on invariants derived from a single instantiation of the protocol. For example, if the protocol involves the parity of nodes, then no single instance can capture all behaviors of the protocol. I4 still requires manual effort, as a developer must inspect a protocol to add additional constraints that reduce the state space to make model checking feasible.

FOL-IC3 [11] infers invariants by searching for logical separators between reachable and invalid states in the protocol. It searches for separators by checking if a separator exists for a fixed number of variables and logical constraints, iteratively increasing the number of possible variables and constraints it considers if it fails. FOL-IC3 provides a strong theoretical guarantee that it can always find the inductive invariant, but does not scale to more complex protocols due to the large space of possible separators that it enumerates and its heavy and repeated use of expensive SMT queries.

We present DistAI (DISTributed protocol Automated learning of Invariants), a fully-automated system for learning inductive invariants for distributed protocols. Like I4, DistAI uses IVy to check invariants, but takes a completely different approach to generating invariants and retrying when invariants fail. We observe that 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. For example, the two-phase commit protocol has an invariant that one node can commit only if all nodes have voted yes, which can be expressed as the following formula:

$$\forall N_1\, N_2.\, \mathit{go\_commit}(N_1) \Rightarrow \mathit{vote\_yes}(N_2). \tag{1}$$

The formula for this invariant only requires two variables, N_1 and N_2, and two relations, $\mathit{go\_commit}(N_1)$ and $\mathit{vote\_yes}(N_2)$, to represent the constraint, but applies to all possible pairs of nodes in an implementation of the protocol regardless of the number of nodes in the implementation. Rather than picking a finite size system from which to generate invariants as in I4, DistAI operates in formula space and picks a finite formula size, with a maximum number of quantified variables (a variable and its quantifier $\forall$ or $\exists$) and literals (a relation such as $\mathit{go\_commit}(N_1)$ in the above example or its negation), for which it enumerates candidate invariants. It then combines the candidate invariants with the desired safety property and feeds them to IVy. If DistAI does not succeed for a given formula size, it increases the formula size and repeats the process until the inductive invariant is found.

Although formula space is finite, enumerating and checking all possible invariants with an SMT solver for even a modest size formula is prohibitively expensive. DistAI limits the set of candidate invariants it feeds IVy to check such that it can still provide a strong theoretical guarantee of finding the inductive invariant while delivering fast performance. DistAI provides this key feature by introducing a novel data-driven approach that uses data from protocol simulations to prune the invariants that are checked to only those that hold for the simulations.

DistAI's data-driven approach starts with the protocol specification used by IVy and automatically converts it into a form it can use to simulate the protocol for various size systems. Protocol simulation simply performs protocol actions by modifying the system state as described in the specification. This generates many raw data samples, where a sample is a snapshot of the system state after an action. Given a formula size, DistAI projects these data samples into subsamples that only involve at most the number of variables allowed by the formula size. For example, if DistAI simulates the two-phase commit protocol for a system with 100 nodes, each data sample would contain the system state for 100 nodes, but each subsample would contain the state of only two of the 100 nodes, assuming a formula size with two variables is being considered as shown in Equation (1).

Using this data, DistAI introduces a novel approach that enumerates only the strongest candidate invariants that hold for all subsamples. An invariant I is stronger than I' if I implies I'. DistAI decomposes the enumeration space of possible invariants based on the number of variables in a formula and starts enumerating smallest formulas first. Any weaker invariants already covered by an already enumerated candidate invariant are skipped. For example, if DistAI has found a candidate invariant $\forall X.p(X)$, it will not enumerate $\forall X.p(X) \vee q(X)$ since the latter is implied by the former. This approach results in fewer candidate invariants being generated, and the candidate invariants generated having smaller formula sizes, but still cover the enumeration space. Feeding these candidate invariants to IVy results in fewer and smaller SMT queries, improving performance. Currently, DistAI only enumerates universal ($\forall$ not $\exists$) invariants.

DistAI does not require sampling to be extensive or complete as the candidate invariants are checked by IVy. If adding the candidate invariants to the desired safety property is inductive, the proof is done; IVy checks if the conjunction of all candidate invariants with the desired safety property is inductive. Otherwise, IVy indicates the invariants that failed, which DistAI then refines. DistAI introduces a novel monotonic invariant refinement approach that we prove is guaranteed to find the right inductive invariant if it can be represented by a given formula size. We prove that the candidate invariants initially generated by DistAI are guaranteed to be stronger than the inductive invariant. As a result, for each candidate invariant that failed, DistAI does not need to discard it, but instead can refine it to a weaker invariant, simply by adding literals. It then tries again by feeding the updated candidate invariants back to IVy. This refinement procedure is strictly monotonic and will converge in a finite number of rounds. If the procedure still fails to find the inductive invariant, DistAI increases the formula size and repeats the whole process of sampling, enumeration, and refinement.

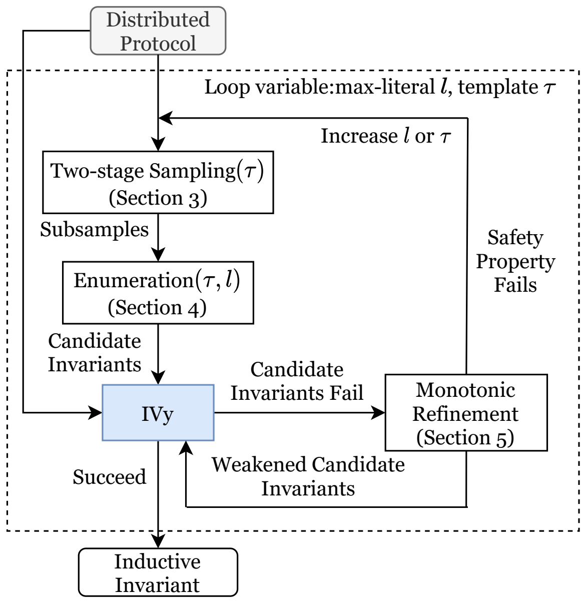

Distributed Protocol
Loop variable:max-literal l, template τ
Increase l or τ
Two-stage Sampling(τ) (Section 3)
Subsamples
Enumeration(τ, l) (Section 4)
Safety Property Fails
Candidate Invariants
Candidate Invariants Fail
IVy
Monotonic Refinement (Section 5)
Succeed
Weakened Candidate Invariants
Inductive Invariant

Figure 1: DistAI workflow.

We prove that if a protocol is verifiable with universal invariants, DistAI is guaranteed to verify it eventually. DistAI operates on formula space, and any invariant formula contains a finite number of variables and literals, so DistAI will converge eventually. Furthermore, DistAI is simple and self-contained, only relying on IVy. It has no other requirements for external components, such as a complex model checker. It also supports protocol abstraction, making it possible to verify protocols that use other protocols, without the need of an executable protocol implementation; only the protocol specification already required to use Ivy is needed.

We demonstrate the effectiveness of DistAI by evaluating it using 14 widely-used distributed protocols in a head-to-head comparison against I4 and FOL-IC3. DistAI outperforms I4 and FOL-IC3 in terms of both the number of protocols for which it finds the inductive invariant and the speed at which it does so. Most protocols take a few seconds and all solved protocols are proven in less than a minute. DistAI succeeds on almost 50% more protocols than either I4 or FOL-IC3. DistAI achieves these results up to an order of magnitude faster than I4 and two orders of magnitude faster than FOL-IC3, without requiring manual effort to add constraints or tune parameters.

2 Overview

Figure 1 illustrates how DistAI works. Starting with a distributed protocol specification for IVy, first, DistAI does two-stage sampling, as discussed in Section 3. It simulates the protocol on different sizes of systems, which we refer to as different size protocol instances, and records the system state as it changes as a sequence of data samples. It then projects the samples into subsamples based on the formula size currently being considered. We express formulas in prenex normal form (PNF), so that the prefix, which we refer to as *an invariant template*, has a maximum number of quantified variables and the matrix has a maximum number of literals. Second, DistAI does enumeration, as discussed in Section 4. It enumerates all strongest candidate invariants that satisfy the subsamples. Third, DistAI feeds the candidate invariants to IVy, which either succeeds with the conjunction of the invariants and the desired safety property as the inductive invariant, or fails and indicates each invariant that does not hold. Fourth, DistAI performs monotonic refinement, as discussed in Section 5. For each candidate invariant that does not hold, DistAI weakens the invariant by adding literals, then feeds the new set of candidate invariants to IVy, repeatedly weakening failed invariants until either it finds the inductive invariant or the safety property itself fails. In the latter case, DistAI increases the formula size by increasing either the maximum number of variables or the maximum number of literals allowed, and repeats the whole process of sampling, enumeration, and refinement.

We use the Ricart-Agrawala protocol [26] as an example of how DistAI works. Figure 2 shows the IVy specification of this classic distributed mutual exclusion protocol, which has five key pieces we use for learning invariants:

1. **Types.** (line 2) Types define different domains of the protocol (e.g., nodes, packets, epochs). The Ricart-Agrawala protocol only has one type, `node`.
2. **Relations.** (lines 4-6) Relations define the state of the protocol, with variables that represent types used as arguments. The Ricart-Agrawala protocol has three relations. For example, relation `holds(N)` has one variable `N` of type node, and indicates if `N` is in the critical section. If the current instance has three nodes N_1,N_2,N_3, then there are three concrete predicates $holds(N_1)$,$holds(N_2)$,$holds(N_3)$ associated with relation *holds*. Each predicate is either `true` or `false` at a certain system state.
3. **Initialization.** (lines 8-12) Initialization defines the initial state of the protocol in terms of its relations. For the Ricart-Agrawala protocol, all relations are initially false.
4. **Actions.** (lines 13-35) Actions define how the protocol may transition from one state to another, modifying the state by setting relations to `true` or `false`. Actions are defined with preconditions using the `require` keyword, which must be satisfied for the protocol to take the action.
5. **Safety Property.** (line 43) The target invariant, defined with logical constraints on the types and relations.

As shown in the specification, a node can send a request for the critical section to another node and can only enter the critical section after it has received replies from all other nodes. When receiving a request, a node delays its reply if it is currently holding the critical section, or if it has requested the critical section and already received a reply from the requester, which indicates an earlier timestamp and a higher priority. The node then sends the reply after it leaves the critical section. The safety property at line 43 asserts that at any time, there is no more than one node holding the critical section. For

```
1  #lang ivy1.7
2  type node
3
4  relation requested(N1:node, N2:node)
5  relation replied(N1:node, N2:node)
6  relation holds(N:node)
7
8  after init {
9      requested(N1, N2) := false;
10     replied(N1, N2) := false;
11     holds(N) := false;
12 }
13 action request(requester: node, responder: node) = {
14     require ~requested(requester, responder);
15     require requester ~= responder;
16     requested(requester, responder) := true;
17 }
18 action reply(requester: node, responder: node) = {
19     require ~replied(requester, responder);
20     require ~holds(responder);
21     require ~replied(responder, requester);
22     require requested(requester, responder);
23     require requester ~= responder;
24     requested(requester, responder) := false;
25     replied(requester, responder) := true;
26 }
27 action enter(requester: node) = {
28     require N ~= requester -> replied(requester, N);
29     holds(requester) := true;
30 }
31 action leave(requester: node) = {
32     require holds(requester);
33     holds(requester) := false;
34     replied(requester, N) := false;
35 }
36
37 export request
38 export reply
39 export enter
40 export leave
41
42 # safety property
43 invariant [safety] holds(N1) & holds(N2) -> N1 = N2
```

Figure 2: Ricart-Agrawala protocol written in IVy. "~" stands for negation. Capitalized variables are implicitly quantified. For example, line 9 means $\forall N_1\ N_2 \in node,\ requested(N_1,N_2) := false$.

simplicity, Figure 2 specifies the protocol without explicit timestamps and only shows one `requested` relation as opposed to separate `request_send` and `request_received` relations which would be part of the real protocol. The safety property of Ricart-Agrawala is not an inductive invariant itself. One needs to add the following two invariants to the safety property so that the resulting conjunction forms an inductive invariant:

$$\forall N_1\ N_2.\ \neg(replied(N_1,N_2) \land replied(N_2,N_1)) \tag{2}$$

$$\forall N_1\ N_2.\ holds(N_1) \land N_1 \neq N_2 \rightarrow replied(N_1,N_2). \tag{3}$$

The first invariant asserts the absence of bidirectional reply, meaning that any two nodes cannot both give the other one a higher priority. The second invariant says that any node holding the critical section must have received replies from all other nodes. DistAI automatically finds the inductive invariant by learning the additional invariants.

Two-stage sampling. To automatically learning the inductive invariant and prove the correctness of the Ricart-Agrawala protocol, DistAI first does two-stage sampling, as shown in Figure 1. It simulates the protocol at different instance sizes and records the system state as a sequence of data samples, each of which presents the values of all the relations. For example, a data sample for an instance size of five nodes (i.e., $n_1, n_2, \cdots, n_5$) using the Ricart-Agrawala protocol consists of 55 boolean values denoting if the following 55 predicates hold or not at the current state:

$$requested(n_1,n_1),\ requested(n_1,n_2), \cdots,\ requested(n_5,n_5)$$
$$replied(n_1,n_1),\ replied(n_1,n_2), \cdots,\ replied(n_5,n_5)$$
$$holds(n_1),\ holds(n_2), \cdots,\ holds(n_5).$$

DistAI chooses a maximum formula size for a candidate invariant, which defines the maximum number of quantified variables that can be used per domain and the maximum number of literals (a predicate or its negation) in the formula. DistAI projects data samples to subsamples, which only contain values of predicates that match the formula size. For example, given a formula with two variables $\{\forall N_1\ N_2\}$, indicating that candidate invariants start with $\forall N_1\ N_2 \cdots$, each subsample only contains the value of predicates related to two assigned nodes.

Enumeration. DistAI then enumerates all strongest candidate invariants that satisfy the subsamples, up to the maximum formula size. Invariants are expressed as formulas in first-order logic. For example, given a maximum formula size of at most two variables and two literals, the following three formulas could be enumerated, assuming they all satisfy the subsamples:

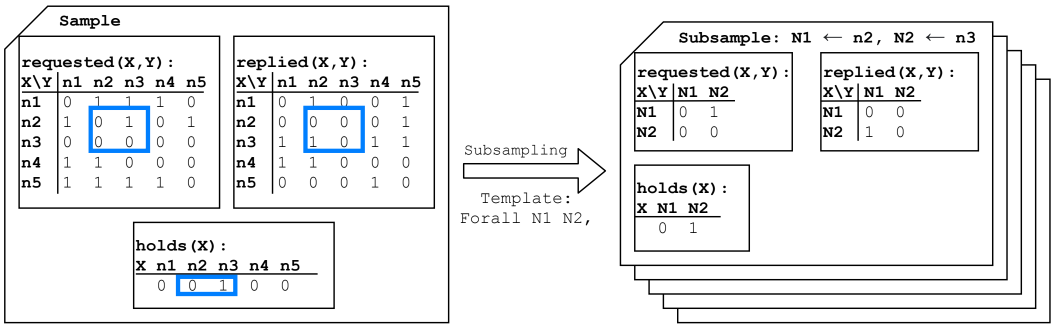
$$\forall N_1 \neq N_2.\ replied(N_1,N_2) \tag{4}$$

$$\forall N_1 \neq N_2.\ replied(N_1,N_2) \lor replied(N_2,N_1) \tag{5}$$

$$\forall N_1 \neq N_2.\ replied(N_1,N_2) \lor \neg holds(N_1). \tag{6}$$

However, DistAI would only generate the first one as a candidate invariant because the first one implies the other two, so the latter two formulas can be skipped. The first formula is the strongest candidate invariant among the three formulas.

Monotonic refinement. DistAI feeds the candidate invariants and the protocol specification to IVy, which runs its SMT solver to check if the conjunction of the invariants with the safety property is an inductive invariant. If the solver passes, DistAI has succeeded. Succeeding means that if the conjunction of the invariants with the safety property holds before a protocol action is taken, each invariant still holds after the action is taken. Otherwise, IVy outputs which candidate invariants failed, and DistAI weakens each failed invariant and tries again with IVy with the candidate invariants, each failed invariant being replaced by weakened invariants with no more variables and literals than the maximum formula size. For the Ricart-Agrawala protocol, IVy shows that the invariant in Equation (4) is incorrect. The invariant is then weakened into Equations (5) and (6), among others, which will then be checked by IVy. Later, IVy will also invalidate Equation (5), and since it has reached the maximum formula size, it will be



requested(X,Y):

X\Y	n1	n2	n3	n4	n5
n1	0	1	1	1	0
n2	1	0	1	0	1
n3	0	0	0	0	0
n4	1	1	0	0	0
n5	1	1	1	1	0

replied(X,Y):

X\Y	n1	n2	n3	n4	n5
n1	0	1	0	0	1
n2	0	0	0	0	1
n3	1	1	0	1	1
n4	1	1	0	0	0
n5	0	0	0	1	0

holds(X):

X	n1	n2	n3	n4	n5
	0	0	1	0	0

requested(X,Y):

X\Y	N1	N2
N1	0	1
N2	0	0

replied(X,Y):

X\Y	N1	N2
N1	0	0
N2	1	0

holds(X):

X	N1	N2
	0	1

Figure 3: The subsampling process. The frame on the left shows a single sample state of a finite instance of the Ricart-Agrawala protocol with five nodes. A single subsample with two quantified variables $\{\forall N_1\ N_2\}$ is generated by mapping the quantified variables to concrete nodes in the finite instance, **n2** and **n3**, and extracting their associated values (shown in blue boxes in the sample frame). $0/1$ stand for false/true.

simply discarded. Equation (6) is never invalidated by IVy and will be part of the inductive invariant in the end. If IVy indicates that the safety property failed, it means that the formula size is not sufficient. DistAI will then increase the formula size by either increasing the maximum number of variables or the maximum number of literals, then re-run the process.

3 Two-Stage Sampling

Obtaining data samples for a distributed protocol requires simulating a finite instance of the protocol and recording the system state on each action. However, invariants are usually composed of quantified variables that impose constraints on all domains of the protocol, not just the specific domains of a finite instance. Therefore, DistAI projects the collected finite state samples into abstract subsamples on quantified variables that also apply to all domains of the protocol and represent potential predicates in the invariant. We refer to these two procedures as *sampling* and *subsampling* respectively, since many abstract subsamples can be generated from a single concrete data sample.

The two-stage sampling has four parameters: the absolute maximum number of instances to consider before terminating (MI), the maximum number of instances to consider before terminating if no further subsamples are generated (MIS), the number of actions to take when simulating a finite instance (MA), and the number of subsamples to generate from one data sample (SD). As we show in Section 6, DistAI's ability to find an inductive invariant is not sensitive to the specific values of these parameters, which are always set to their defaults of 1000, 20, 50, and 3, respectively.

Sampling. DistAI first translates the protocol from IVy into Python, with relations simulated by multidimensional arrays, and actions simulated by Python functions. This allows DistAI to efficiently simulate the protocol. The translation is not in the trusted computing base since learned invariants are eventually validated by IVy.

DistAI then simulates the protocol in Python from different valid initial states on randomly chosen finite instances of the protocol. DistAI randomly chooses an instance size from some range of sizes using a simple discrete uniform distribution. For each domain T (e.g., node), a protocol typically has some minimum instance size to function well, which we refer to as N^T_{min}. In practice, the minimum instance size N^T_{min} is determined as the maximum number of variables of type T in any relation; a protocol will not function well if its relations have variables that cannot be mapped to the instance size. For example, for a protocol with two relations $p(n_1:T_1, n_2:T_1, m_1:T_2)$ and $q(m_1:T_2)$, we have $N^{T_1}_{min} = \max(2,0) = 2$ and $N^{T_2}_{min} = \max(1,1) = 1$. The probability for choosing a given domain size N^T is then:

$$Pr[N^T] = 1/w \ \ (N^T_{min} \leq N^T < N^T_{min} + w)$$

DistAI uses $w = 4$ by default. This allows sampling from multiple instance sizes, but limits the instance sizes to within w of the minimum instance size for performance, as larger instances take more time to simulate.

For each valid initial state, DistAI simulates the protocol by performing MA number of actions. Since the distributed protocols are nondeterministic with regard to the next action taken (e.g., we do not know which node will send the next request or reply), multiple runs from the same initial state will result in different samples. Given an initial state s_0, DistAI uses the simple method formalized in Algorithm 1, to simulate a protocol, which randomly chooses an action from an action pool (line 6) with randomly chosen arguments from an argument pool (line 9) that satisfy the precondition (line 10). It then performs the action, records the new system state, and repeats the process (line 16-17). An action is removed from the action pool once its argument pool is exhausted, and the protocol terminates if the action pool is exhausted. Since some protocols may never terminate, MA defines an upper bound on the number of actions performed. Once the simulation completes, the set of reached states S is returned.

For example, for the Ricart-Agrawala protocol, during each iteration of the algorithm, DistAI first randomly selects one of the four possible actions: `request`, `reply`, `enter`, and `leave`. If `request` is selected, DistAI then randomly chooses the nodes for its two arguments, `requester` and `responder`. However, not every pair of nodes are valid arguments as the two nodes must satisfy lines 14-15, the two preconditions to legitimately trigger the `request` action under the protocol. If the current

Algorithm 1 Stochastic Sampling Algorithm.

Input: Protocol $\mathcal{P}$ with actions $\mathcal{A}$. Initial state s_0
Output: A simulation trace, represented by a set of states S

```
1:  $S := \{s_0\}, s := s_0$
2:  for $iter := 1$ to $MA$ do
3:      $action\_pool := \mathcal{A}$
4:      $action\_found := false$
5:      while $\neg action\_found \wedge |action\_pool| > 0$ do
6:          $action :=$ select_random($action\_pool$)
7:          $args\_pool :=$ enum_arguments($s$,$a$)
8:          while $|args\_pool| > 0$ do
9:              $args :=$ select_random($args\_pool$)
10:             if precondition_holds($\mathcal{P}$,$s$,$action$,$args$) then
11:                 $action\_found := true$
12:                 break
13:             $args\_pool := args\_pool \setminus \{args\}$
14:         $action\_pool := action\_pool \setminus \{action\}$
15:     if $action\_found$ then
16:         $s$:=execute_protocol($\mathcal{P}$,$s$,$a$,$args$)
17:         $S$:=$S \cup \{s\}$
18:     else
19:         break
20: return $S$
```

⟨`requester`,`responder`⟩ pair violates the precondition, DistAI removes it from the argument pool and randomly selects another one. This repeats until a valid pair of arguments is found or the pool is exhausted, in which case DistAI removes `request` from the action pool and selects another action.

After each iteration, DistAI logs the current system state, represented by the value of all the relations. In Figure 2, that is the value of predicates $requested(N_1,N_2)$, $replied(N_1,N_2)$, and $holds(N_1)$ for all N_1 and N_2. Figure 3 shows a sample of the Ricart-Agrawala algorithm for an instance size of five nodes in the left frame. The *requested* and *replied* relations each take two nodes as arguments, so their samples record the relations for all possible pairs of nodes, resulting in 25 boolean values each. The *holds* relation only takes a single node as argument, so five boolean values are recorded, one for each of the five nodes in the protocol instantiation.

DistAI can also simulate protocols calling other protocols, even when the protocol being called is a blackbox. When a protocol calls another protocol through a blackbox interface, described by a specification without a concrete implementation, DistAI treats it as an action with nondeterministic behavior. If the action is selected with arguments that satisfy its preconditions, DistAI selects randomly updated states that satisfy its postconditions as the execution result.

Our simple stochastic sampling procedure, while very efficient, may not achieve high coverage and can leave corner cases uncovered. More sophisticated techniques [1, 25, 30] can be applied to improve coverage for complex protocols with sparse inputs and difficult to reach states. However, the correctness of learned invariants is guarded by the Z3 SMT solver used by IVy. If the samples are incomplete and the invariants fail the SMT check, DistAI will iteratively refine the invariants until they are correct, as discussed in Section 5.

Subsampling. The data samples from protocol simulation may be of all different lengths depending on the instance size used. We want to map the concrete samples from simulation to an invariant template, the small set of quantified variables that may appear in the invariant, denoted by τ. Given a set of data samples and an invariant template, DistAI applies a subsampling procedure translating the variable length data samples to fixed length vectors, which we call *subsamples*. Formally, a subsample corresponds to an assignment α of the variables in τ and contains the values of relations given the assignment to the template. Subsamples taken with an invariant template with variables $V_1,...,V_n$ can then be used to learn invariants (denoted I) on those variables:

$$\tau = \{\forall V_1 ... V_n\} \qquad \forall V_1 ... V_n.\, I(V_1,...,V_n).$$

For example, in the Ricart-Agrawala protocol, the relations *requested* and *replied* each operate on two nodes, so the initial template is $\tau = \{\forall N_1\ N_2\}$. Under this template, there are only 10 predicates that may appear in an invariant formula, namely:

$$requested(N_1,N_1),\ requested(N_1,N_2),\ requested(N_2,N_1),\ requested(N_2,N_2),\ replied(N_1,N_1),\ replied(N_1,N_2),\ replied(N_2,N_1),\ replied(N_2,N_2),\ holds(N_1),\ holds(N_2).$$

For this template, a 5-node data sample can induce $5*4=20$ subsamples, by first assigning one node X_1 for N_1 and then another node X_2 for N_2 , as illustrated in Figure 3. Since there are 10 predicates, each subsample has 10 possible boolean values, so one data sample results in $20 \times 10 = 200$ boolean values.

Enumerating all valid subsamples from each sample can be computationally undesirable, especially for multi-domain protocols. If we add a new domain `msg`, and let the template be $\{\forall\ N_1\ N_2 \in$ `node`, $M_1\ M_2 \in$ `msg`$\}$, then a sample with five nodes and 10 messages will induce 1,800 subsamples. Therefore, DistAI randomly chooses *SD* valid subsamples from each sample. Two-stage sampling terminates when *MI* instances have been simulated, or no new subsample is found after simulating *MIS* consecutive instances of the protocol. The subsamples are then deduplicated and passed on to invariant enumeration.

Although DistAI's sampling has several parameters, they do not need to be manually tuned to find inductive invariants. We use the default values for all protocols. For example, when *MA* or *SD* become larger, each simulation round will take longer, but fewer rounds will be required to converge. Similarly, a small MI/MIS may stop two-stage sampling prematurely, but the missing states will be resolved later by monotonic refinement. The parameters do not affect whether DistAI finds inductive invariants, only how fast it finds them. Sampling is useful simply as a performance optimization that reduces the number of SMT queries required during refinement.

4 Candidate Invariant Enumeration

Algorithm 2 shows DistAI's enumeration-based algorithm to generate candidate invariants from the subsamples obtained in Section 3. To reduce the number of candidate invariants required for covering the invariant space and reduce the maximum number of literals needed for finding the inductive invariant, we partition the invariant space into multiple regions, each represented by a constrained template called a *subtemplate*. We then enumerate all possible invariants in each region (i.e., under each subtemplate), and retain candidate invariants that hold for the collected subsamples.

Template decomposition. Before enumerating candidates invariants, we decompose templates into subtemplates that incorporate additional constraints (line 1). A template with N variables in the same domain will be split into N subtemplates which have from 1 to N variables. A subtemplate with more variables is said to be larger than a subtemplate with fewer variables. For example, a template $\tau = \{\forall N_1\ N_2\}$ will be split into two subtemplates, $\tau_1 = \{\forall N_1\ N_2.\ N_1 = N_2\} = \{\forall N_1\}$ and $\tau_2 = \{\forall N_1\ N_2.\ N_1 \neq N_2\}$, abbreviated as $\{\forall N_1 \neq N_2\}$, with τ_2 being larger than τ_1. All operations that use subtemplates in Algorithm 2 traverse them from smallest to largest (lines 2 & 5).

This subtemplate optimization reduces the cost of enumeration in two ways. First, subtemplates reduce the number of candidates that need to be enumerated due to symmetry. For example, for the Ricart-Agrawala protocol, when using template τ, both of the following invariants will be enumerated:

$$\forall N_1\ N_2.\ \neg replied(N_1,N_1) \qquad \forall N_1\ N_2.\ \neg replied(N_2,N_2).$$

On the other hand, when using the subtemplate τ_1, the equivalent enumeration would only result in one candidate:

$$\forall N_1.\ \neg replied(N_1,N_1). \tag{7}$$

Furthermore, DistAI will project Equation (7) using τ_2 to the following candidate invariants:

$$\forall N_1 \neq N_2.\ \neg replied(N_1,N_1) \quad \forall N_1 \neq N_2.\ \neg replied(N_2,N_2),$$

which are then marked as validated using τ_1, avoiding further redundant validations. We refer to this as invariant projection.

Second, subtemplates can reduce the maximum number of literals in the invariant formula. For example, one invariant of the Ricart-Agrawala protocol (Equation 3) can be rewritten as:

$$\forall N_1\ N_2.\ \neg(N_1 \neq N_2) \vee \neg holds(N_1) \vee replied(N_1,N_2) \tag{8}$$

This is a disjunction of three literals under the full template τ. However, using subtemplate $\tau_2 = \{\forall N_1 \neq N_2\}$, an equivalent

Algorithm 2 Invariant Enumeration Algorithm.

Input: Template τ, subsample table ST, max-literal l
Output: A set of invariants I^*

```
 1: subtemplates := decompose_templates($\tau$)
 2: for $\tau' \in$ traverse(subtemplates) do
 3:     proj_table[$\tau'$] := $ST_{|\tau'}$
 4:     $I[\tau'] := \emptyset$
 5: for $\tau' \in$ traverse(subtemplates) do
 6:     predicates := proj_table[$\tau'$].header
 7:     $\mathcal{P}_\tau$ := predicates $\cup \{\neg p | p \in$ predicates$\}$
 8:     for $n := 1$ to $l$ do
 9:         for $inv \in$ combinations($\mathcal{P}_\tau$, $n$) do
10:             if check_subset_exists($inv$, $I[\tau']$) then
11:                 continue
12:             if check_inv_holds($inv$, proj_table[$\tau'$]) then
13:                 $I[\tau'] := I[\tau'] \cup \{inv\}$
14:     for $\tau'_{succ} \in successors(\tau')$ do
15:         for $inv \in I[\tau']$ do
16:             $I[\tau'_{succ}] := I[\tau'_{succ}] \cup$ proj_inv($inv$,$\tau'$,$\tau'_{succ}$)
17: $I^*$ := $\{(\tau' : inv) \mid \tau' \in$ subtemplates, $inv \in$
    $I[\tau']$, $inv$ was checked against subsamples$\}$
```

form of this invariant can be learned using a formula size with a maximum of two literals:

$$\forall N_1 \neq N_2.\ \neg holds(N_1) \vee replied(N_1,N_2)$$

We can denote an invariant as $\tau : inv$, where τ is the subtemplate under which the invariant formula inv is found and inv is expressed as a disjunction of literals. In this example, we effectively can denote the same invariant using subtemplate τ_2 as $\tau_2 : inv'$, where one literal that was previously part of inv is no longer part of inv' because it is now a part of τ_2. Because DistAI operates in formula space and the time complexity of enumeration is exponential in the maximum number of literals, such a small reduction in the number of literals can have a significant impact on the overall cost of enumeration.

Subtemplates can also reduce the maximum number of literals by exploiting another form of symmetry. If there is a total order on a domain, such as with node identifiers, we will further assign an order on the variables in the subtemplate and strengthen $\{\forall N_1 \neq N_2\}$ into $\{\forall N_1 < N_2\}$ Because of symmetry, invariant formulas with $\{\forall N_1 < N_2\}$ are equivalent to those with $\{\forall N_2 < N_1\}$, so we do not need to enumerate the latter once we have done the former. This is useful since inductive invariants often contain comparisons between variables in a domain with a total order. For example, this optimization helps reduce the maximum number of literals required from six to four for the database chain replication protocol evaluated in Section 6.

Subtemplates work with multiple domains as well. Consider a protocol with two domains, T_1 and T_2, where T_1 defines a total order, and the template τ is $\{\forall X_1\ X_2\ X_3 \in T_1, Y_1\ Y_2 \in T_2\}$. After

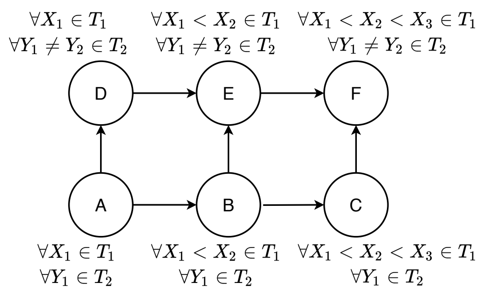

$\forall X_1 \in T_1$ $\forall Y_1 \neq Y_2 \in T_2$
$\forall X_1 < X_2 \in T_1$ $\forall Y_1 \neq Y_2 \in T_2$
$\forall X_1 < X_2 < X_3 \in T_1$ $\forall Y_1 \neq Y_2 \in T_2$
D
E
F
A
B
C
$\forall X_1 \in T_1$ $\forall Y_1 \in T_2$
$\forall X_1 < X_2 \in T_1$ $\forall Y_1 \in T_2$
$\forall X_1 < X_2 < X_3 \in T_1$ $\forall Y_1 \in T_2$

Figure 4: Dependency relations between the six subtemplates derived from the template $\{\forall X_1\ X_2\ X_3 \in T_1,\ Y_1\ Y_2 \in T_2\}$.

template decomposition we get six subtemplates, as shown in Figure 4. For multiple domains, there may not exist a total ordering of all subtemplates from smallest to largest, so the only requirement for the order of traversal of subtemplates is that the quantified variables in each subtemplate are not a subset of a prior one, such that formulas can always be validated with the smallest possible subtemplate. In Figure 4, $A \to B \to C \to D \to E \to F$ is a valid traversal order, while $A \to D \to E \to B \to C \to F$ would be invalid because the quantified variables $\{X_1, X_2, Y_1\}$ for subtemplate B are a subset of $\{X_1, X_2, Y_1, Y_2\}$ of subtemplate E. We follow graph terminology and call subtemplates B and D the *successors* of A, and A the *predecessor* of B and D.

Subtemplate projection. Because DistAI uses subtemplates for candidate enumeration instead of templates, the full subsamples of the invariant template need to be projected onto each subtemplate (line 3). This is done similarly to how data samples are projected onto full subsamples using an invariant template, as discussed in Section 3, except in this case, full subsamples are projected onto subsamples using a subtemplate. For example, for this multi-domain protocol, when projecting full subsamples to subtemplate $\{\forall X_1 < X_2 \in T_1,\ \forall Y_1 \in T_2\}$ (node B in Figure 4), there are six possible variable mappings: $\{X_1 \to X_1, X_2 \to X_2, Y_1 \to Y_1\}, \ldots, \{X_1 \to X_2, X_2 \to X_3, Y_1 \to Y_2\}$. Note that the total order on T_1 needs to be preserved, otherwise there would be 12 possible mappings. Similarly, going to back the Ricart-Agrawala protocol example, when projecting full subsamples of the invariant template τ to subtemplate τ_1, there are two possible variable mappings: $\{N_1 \to N_1, N_1 \to N_2\}$.

Subtemplate candidate enumeration. DistAI enumerates and checks all possible candidates for each subtemplate τ' (lines 6-13). Each subtemplate τ' has a certain number of predicates m. For example, for the Ricart-Agrawala protocol using template τ_1, there are three predicates: $requested(N_1,\ N_1)$, $replied(N_1,\ N_1)$, $holds(N_1)$. DistAI adds the m predicates $p_1,\ p_2,\ \ldots,\ p_m$ and their negations $\neg p_1, \neg p_2, \ldots, \neg p_m$ to the literal pool $\mathcal{P}_\tau$ (line 7).

Given a formula size with the maximum number of literals l, DistAI enumerates all subsets of $\mathcal{P}_\tau$ with size at most l as candidate invariants. For example, if $m = 3$ and $l = 1$, there would be six candidate invariants: $p_1, \neg p_1, p_2, \neg p_2, p_3, \neg p_3$. By default, DistAI initially sets $l = 3$, and iteratively increases it later in the refinement process (see Section 5). We only consider invariants in the form of disjunctions of literals since invariants with conjunctions can simply be split into multiple invariants. If a candidate invariant C includes both a predicate and its negation, it will be discarded. If not, DistAI checks the validity of C against the subsamples. If C is satisfied by all subsamples for the subtemplate, C is added to the set of generated invariants, which we refer to as the invariant set.

DistAI exploits symmetry to prune the candidate enumeration space. Whenever an invariant is learned, we permute the quantified variables with the same type and emit equivalent candidates without needing to check if they are satisfied by the subsamples. For example, under the subtemplate $\{\forall\ X \neq Y \in T_1,\ A \neq B \in T_2\}$, if $p(X,Y) \vee \neg q(Y) \vee r(X,A,B)$ is an invariant, then $p(Y,X) \vee \neg q(X) \vee r(Y,B,A)$, along with two other formulas, are also invariants.

Enumeration is ordered by the number of literals in the candidate invariants, and any candidate that is weaker than an invariant already added to the invariant set is skipped (lines 8-11). For example, if we already know $p \vee \neg q$ is an invariant, then for any predicate r, $p \vee \neg q \vee r$ is guaranteed to be a valid but weaker invariant, and can be skipped in the enumeration. Based on Figure 3, applying this enumeration procedure to the Ricart-Agrawala protocol with subtemplate $\{\forall N_1\}$ and $l = 2$ results in the following two generated invariants:

$$\neg requested(N_1, N_1) \qquad \neg replied(N_1, N_1)$$

Invariant projection. After finding all candidates on one subtemplate, DistAI calculates the projection of the candidates on each successor, then propagates the projection and moves on to enumerate the next subtemplate (line 14-16). This reduces the cost of validating candidates using larger subtemplates against their subsamples. For example, for the Ricart-Agrawala protocol, suppose we have learned two invariants $\neg requested(N_1, N_1)$ and $\neg replied(N_1, N_1)$ under subtemplate $\tau_1 = \{\forall N_1\}$. Before enumerating candidates under subtemplate $\tau_2 = \{\forall N_1\ N_2\}$, we know the following four candidates must hold under τ_2 because they are projections of the learned invariants under the simpler template τ_1:

$$\begin{array}{ll} \neg requested(N_1, N_1) & \neg replied(N_1, N_1) \\ \neg requested(N_2, N_2) & \neg replied(N_2, N_2) \end{array}$$

As a result, these four candidates can simply be added to the invariant set under τ_2 without enumerating and validating them against any subsamples (line 16). Any weaker candidates will also be skipped, further reducing the cost of enumeration. For example, $\neg requested(N_1, N_1) \vee holds(N_1)$ can be skipped since it is weaker than $\neg requested(N_1, N_1)$.

Strongest possible invariant set. Finally, after traversing all subtemplates, DistAI unions together the subtemplate invariant sets to form the initial set of generated invariants (line 17) which will be fed to IVy. Since all possible candidate invariants have been considered for each subtemplate, we can prove that, for any invariant inv (in the form of disjunctions of literals) under template τ with a maximum number of literals of no more than l, there must exist an invariant inv' in the constructed invariant set such that $inv' \Rightarrow inv$. General invariants can be converted into conjunctive normal form (CNF) and then split into multiple invariants in the form of disjunctions of literals. Thus, the initial invariant set for a subtemplate is a *strongest possible* invariant set that is guaranteed to be at least as strong as the inductive invariant if there are no more than τ variables, also known as quantifiers, and l literals.

Intuitively, since each subtemplate provides the strongest possible invariant set, the invariant checked by IVy, which is constructed using the conjunction of all invariants across the union of subtemplate invariant sets, should also be the strongest with regard to the subsamples. In practice, when unioning the invariant sets, we can exclude invariants generated by projection from predecessors because they can be implied by their original counterparts. We formalize this in Theorem 1:

Theorem 1. *Let I^* be the output of Algorithm 2. For any invariant set I under template τ with a maximum number of literals of no more than l, if I is satisfied by every subsample, then $I^* \Rightarrow I$.*

Proof. First we consider a variant of Algorithm 2 where Line 17 does not exclude invariants generated by projection. We prove this by contradiction. Suppose there exists I under template τ with a maximum number of literals of no more than l, I is satisfied by every subsample and $I^* \not\Rightarrow I$. Consider any invariant $\tau' : inv \in I$ but $\tau' : inv \notin I^*$. Recall each individual invariant is a disjunction of literals, assuming CNF. Since I is satisfied by every subsample, $\tau' : inv$ is also satisfied by every subsample. If we reach lines 12-13 in Algorithm 2, it will be added to the invariant set. The only possibility of $\tau' : inv \notin I^*$ is that the branch condition at line 10 evaluates to $true$. However, this indicates that a subset of inv is already in the invariant set. The subset of inv implies inv (e.g., $p \vee q \Rightarrow p \vee q \vee r$). So we still have $I^* \Rightarrow \tau' : inv$. To conclude, every $\tau' : inv \in I$ but $\tau' : inv \notin I^*$ can be implied by I^*, which is a contradiction to $I^* \not\Rightarrow I$.

Now we exclude invariants generated by projection and get a new I^*_{new}. From lines 15-16, every excluded invariant can be implied by another invariant in its predecessor subtemplate, so we can show $I^* \Leftrightarrow I^*_{new}$, thus completing the proof. □

Constants and function symbols. Although the discussion above assumes a literal can only be a predicate or its negation, DistAI also supports constants and function symbols as literals. For example, given a template $\{\forall X\ Y \in T\}$ and a constant $c \in T$, DistAI considers $X = c$ and $Y = c$ as two independent predicates and reasons about them like any other predicate. As another example, given a template $\{\forall X_1\ X_2 \in T_1, Y_1 \in T_2\}$ and a function $f : T_1 \rightarrow T_2$, DistAI can introduce $Y_2 = f(X_1)$,$Y_3 = f(X_2)$ and treat Y_2,Y_3 as variables like Y_1.

5 Monotonic Invariant Refinement

When DistAI feeds the enumerated invariants to IVy, IVy may find that the conjunction of the invariants and the safety property are not inductive and return a list of invariants that failed. This is likely to happen at least for the initial invariants that DistAI enumerates as its sampling is not guaranteed to be complete. Because sampling is not complete and is primarily to improve performance, DistAI may generate invariants that would not hold if sampling was done for more protocol instances. In general, when IVy indicates that an invariant fails, it is difficult to know whether the solution is to weaken or strengthen the invariant. Prior work uses different methods to evade this challenge but gives no fundamental solution [19,35].

DistAI provides a simple and clean solution to this problem by starting with the strongest possible invariants and ensuring that the invariants remains the strongest possible ones throughout the refinement process. For each invariant that fails, which we refer to as a broken invariant, DistAI applies *mimimum weakening* to the invariant. The candidate invariant space becomes strictly smaller after each failure. DistAI ensures that the conjunction of the weakened invariants will remain stronger than the eventual invariants that must be added to the safety property to make it inductive, if it is expressible under the current template τ and maximum number of literals l. The overall process is guaranteed to converge to find the inductive invariant.

Algorithm 3 Minimum Weakening Algorithm.

Input: Invariant set $I[\tau']$ for each subtemplate τ', and the broken invariant $\tau'_0 : inv$
Output: Updated invariant set $I[\tau']$ for each subtemplate τ'

```
1:  $I[\tau'_0] := I[\tau'_0] \setminus \{inv\}$
2:  if $inv.length < l$ then
3:      for $literal \in$ valid_literals($\tau'_0$) do
4:          if $literal \notin inv$ then
5:              $new\_inv := inv \cup \{literal\}$
6:              if $\neg$ check_subset_exists($new\_inv$, $I[\tau'_0]$) then
7:                  $I[\tau'_0] := I[\tau'_0] \cup \{new\_inv\}$
8:  for $\tau'_{succ} \in successors(\tau'_0)$ do
9:      $new\_invs$:=proj_inv($inv$,$\tau'_0$,$\tau'_{succ}$)
10:     for $new\_inv \in new\_invs$ do
11:         $I[\tau'_{succ}] := I[\tau'_{succ}] \cup \{new\_inv\}$
```

Algorithm 3 shows the minimum weakening algorithm used, given an initial invariant set and a broken invariant. We denote an invariant as $\tau' : inv$, where τ' is the subtemplate under which inv is found and inv is the invariant expressed as a disjunction of literals. The algorithm consists of three steps. First, DistAI removes the broken invariant from the initial invariant set. When IVy returns that $\tau'_0 : inv$ fails, DistAI removes $\tau' : inv$ from the invariant set that was initially passed to IVy (line 1).

Second, DistAI finds all weakened versions of the broken invariant and adds them back to the invariant set. A weakened version of $\tau' : inv$ is created by add one more literal via disjunction to inv (lines 2-7). For example, suppose $inv = p \vee \neg q$ is rejected by IVy. Recall that during the invariant enumeration, since $p \vee \neg q$ was considered as an invariant, for all literals r, the candidate $p \vee \neg q \vee r$ would be skipped. Now, for any literal r, $p \vee \neg q \vee r$ becomes a meaningful invariant. DistAI updates the invariant set by adding the weakened invariants back to the invariant set as long as they can not be implied by some other invariant that is already in the invariant set (e.g., $p \vee r$). If the broken invariant has reached the maximum number of literals, this second step will be skipped.

Third, DistAI projects the broken invariant to higher subtemplates, and adds all such projections. For each successor τ'_{succ} of τ'_0, DistAI adds all the projections of *inv* on τ'_{succ} to the invariant set (line 8-11). To see why this is necessary, consider the following candidate invariant in some leader election protocol:

$$\forall X \in T.\ \neg leader(X). \tag{9}$$

This asserts no one can be a leader. This invariant may fail in IVy because the SMT solver observes a system state $\{leader(i_1), \neg leader(i_2), i_2 < i_1\}$ (suppose T has total order). Recall that DistAI uses a traversal order to enumerate invariants under different subtemplates, and the invariants from smaller subtemplates will be projected to larger subtemplates to avoid repeated enumeration. So previously, the following two candidate invariants, under a larger subtemplate $\{\forall X < Y \in T\}$, were skipped because they could be implied by Invariant (9).

$$\forall X < Y \in T.\ \neg leader(X) \tag{10}$$

$$\forall X < Y \in T.\ \neg leader(Y). \tag{11}$$

But now, after Invariant (9) is invalidated and removed, we need to reconsider Invariants (10) and (11) and add them to the candidate invariant set. We validate the new invariants using IVy again. If successful, DistAI outputs the current invariant set as the inductive invariant, otherwise we will enter the next refinement round. In this case, if the distributed protocol has the property that only the greatest user can be the leader, then Invariant (11) will be invalidated in a later round, while Invariant (10) is likely to be correct and remain valid to the end.

The three-step minimum weakening procedure guarantees after any number of refinement rounds, the invariant set is always a strongest possible one that is satisfied by *all* the subsamples. This "strongest possible" property implies that throughout refinement, the invariant set is always stronger than the correct invariant set required for an inductive invariant, so whenever an invariant fails, we should always weaken the broken invariant. The guarantee can be formally stated as:

Theorem 2. *Let I^* be the invariant set after n refinement rounds, and $B_n = \{\tau'_1 : inv_1, \tau'_2 : inv_2, ..., \tau'_n : inv_n\}$ be the broken invariants in each round. For any invariant set I under template τ with no more than l literals, if I is satisfied by every subsample, and does not imply any broken invariant in B_n, then $I^* \Rightarrow I$.*

Proof. We prove this by induction on the number of rounds. The base case is simple. In Round 0, there is no broken invariant, and the statement degenerates to Theorem 1. Now we focus on the induction case. Suppose after k refinement rounds, we get invariant set I^*_k. For any invariant set I under template τ with no more than l literals, if I is satisfied by every subsample, and does not imply any broken invariant in B_k, then $I^*_k \Rightarrow I$.

Now we come to round $k+1$. We prove by contradiction. Suppose we have an invariant set I under template τ with no more than l literals such that 1) I is satisfied by every subsample, 2) I does not imply any broken invariant in B_{k+1}, and 3) $I^*_{k+1} \not\Rightarrow I$. Consider any invariant $\tau' : inv$ such that $I \Rightarrow \tau' : inv$ but $I^*_{k+1} \not\Rightarrow \tau' : inv$. From the induction hypothesis, we know $I^*_k \Rightarrow \tau' : inv$. Let $\tau'_{k+1} : inv_{k+1}$ be the invalidated invariant in round $k+1$. From the algorithm, $\tau'_{k+1} : inv_{k+1}$ is the only removed invariant in this round. Since each invariant is a disjunction of literals, we can show $\tau'_{k+1} : inv_{k+1} \Rightarrow \tau' : inv$. In other words, the hypothetical "missing" invariant must be implied by the removed invariant. We further know either *inv* includes more literals than *inv*, or τ' includes more quantified variables not in τ'_{k+1}, or both. Otherwise we have $\tau' : inv \Rightarrow \tau'_{k+1} : inv_{k+1}$. Then $\tau' : inv = \tau'_{k+1} : inv_{k+1}$, a contradiction to $I \Rightarrow \tau' : inv$ and I does not imply the broken invariant $\tau'_{k+1} : inv_{k+1}$.

Now we consider the two cases separately. 1) *inv* includes a literal p not in inv_{k+1}. Consider the formula $F = \tau_{k+1} : inv_{k+1} \vee p$. From $\tau'_{k+1} : inv_{k+1} \Rightarrow \tau' : inv$, we can show $F \Rightarrow \tau' : inv$. However, F is added to the new invariant set I^*_{k+1} unless it can be implied by existing invariants (Line 3-7 in Algorithm 3). So we have $I^*_{k+1} \Rightarrow F \Rightarrow \tau' : inv$. 2) τ' includes a quantified variable X not in τ'_{k+1}. Consider the formula $G = \tau'' : inv_{k+1}$, where τ'' is τ' extended with X. Again, from $\tau'_{k+1} : inv_{k+1} \Rightarrow \tau' : inv$, we can show $G \Rightarrow \tau' : inv$. However, G is added to the new invariant set I^*_{k+1}(Line 8-11 in Algorithm 3). So we have $I^*_{k+1} \Rightarrow G \Rightarrow \tau' : inv$. In both 1) and 2), we reach $I^*_{k+1} \Rightarrow \tau' : inv$, which means the "missing" invariant is already implied by the existing invariant set output by the algorithm, a contradiction. □

Intuitively, Theorem 2 ensures that starting from a too strong invariant set, the minimum weakening steps never over-weaken the invariants and "bypass" the correct invariants in between. Combined with Theorem 1, which guarantees that monotonic refinement indeed starts from the strongest invariant set, we have the following corollary:

Corollary 1. *If there exists a correct invariant set expressible with template τ and maximum number of literals l, then the refinement procedure will terminate with one such invariant set within a finite number of rounds, otherwise the refinement procedure will terminate with a broken safety property.*

Sometimes, the weakened versions of a broken invariant are all discarded in the end. Then, minimum weakening provides no benefits versus just removing the broken invariants. In practice, DistAI first applies only the first step of minimum weakening — removing the broken invariants. Then if failed, DistAI applies refinement again with the first and second step. If failed again, DistAI applies the standard three-step minimum weakening in Algorithm 3. This practice optimizes performance when the weakened versions of a broken invariant are all discarded while maintaining Theorem 2 and Corollary 1.

If available, DistAI can use counterexamples to check weakened invariants, only adding them if they satisfy the counterexamples. However, DistAI's refinement procedure currently does not use them because obtaining counterexamples from IVy for an entire invariant set is extremely inefficient. When IVy is configured to return a counterexample, it halts early and returns the counterexample once it identifies the first broken invariant in a set. Using IVy in this configuration would force DistAI to weaken broken invariants one at a time and perform many redundant SMT checks of the invariant set through IVy, instead of weakening all failed invariants at once between each IVy call.

Convergence and Feedback loop. Since the invariant set is weakened after each refinement round, we can prove that the refinement procedure terminates in a finite number of rounds, resulting in an inductive invariant set.

If the safety property has never been violated during the refinement process, the resulting set is the correct inductive set and can derive the desired safety property. If, at any point, the safety property is violated and needs to be weakened (when all other candidates are weak enough), it means that the correct invariant set cannot be expressed under the current formula size, with its per-domain template size and maximum number of literals. DistAI will then increase the formula size, increasing the per-domain template size or the maximum number of literals, and relearn the invariants. By default, DistAI increases the formula size by alternating between increasing the maximum number of literals or increasing the template size, the latter by increasing the number of quantified variables for each domain in the template. For example, in Figure 4, i) we first increase the maximum number of literals by one, ii) if it fails, increase the template size by adding a new variable in type T_1, iii) if fails again, add another new variable in T_2, iv) and if still fails, increase the maximum number of literals by one again.

After increasing the formula size, we redo sampling, enumeration, and refinement. Since any invariant contains a finite number of quantified variables and a finite number of literals, the feedback loop will eventually reach a template and literal size large enough to express the correct invariant set if one exists. Once a sufficient formula size is reached, Corollary 1 guarantees that a correct invariant set will be generated. Therefore, DistAI provides the following end-to-end convergence guarantee:

Theorem 3. *If the safety property of a protocol is provable with a $\exists$-free invariant set, then DistAI will terminate with one such invariant set in finite time.*

Theorem 3 guarantees conditional convergence of DistAI. However, if the safety property does not hold for the protocol or existential quantifiers are necessary to prove it correct, DistAI may continue in the feedback loop forever.

6 Evaluation

To demonstrate its effectiveness at determining inductive invariants, we implemented and evaluate DistAI on a collection of 14 distributed protocols, including all 7 protocols previously evaluated with I4 [18]. The implementation consists of 1.6K lines of Python code for protocol simulation and sampling and 1.6K lines of C++ code for enumeration and refinement. For comparison, we also evaluated I4 and FOL-IC3, in both cases using the implementations created by the original authors. 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. Table 1 shows the results for each protocol, including the number of domains and relations for each protocol as indicators of protocol complexity.

DistAI outperforms both I4 and FOL-IC3 in terms of the number of protocols for which it infers the correct invariants. DistAI automatically infers the correct invariants for 13 out of the 14 protocols, only failing for Paxos, on which both I4 and FOL-IC3 also fail. I4 only solves 9 protocols, and FOL-IC3 only solves 3 protocols using its default setting, which searches over all first-order logic formulas, but improves to solving 9 protocols if an option is enabled that limits the search space to only $\forall$ quantifiers. Each approach was allowed to run for an entire week, 168 hours, per protocol before timing out, more than two orders of magnitude longer than the worst runtime reported in Table 1.

DistAI and I4 only time out trying to solve Paxos, but FOL-IC3 times out on many protocols. This is because DistAI only uses the SMT solver for validating rather than generating invariants, I4 uses a model checker to generate invariants only for a specific, small instance, while FOL-IC3 invokes the SMT solver to generate invariants for the general protocol, multiple times for each invariant, which is undecidable in general and very expensive in practice. FOL-IC3 performs worse with the default setting since the formula search space is larger and the SMT solver performs worse for formulas with existential quantifiers. In fact, FOL-IC3 fails for database chain replication, decentralized lock, and distributed lock with the default setting because Z3, the underlying SMT solver, fails and reports `unknown`, indicating that the formula generated by FOL-IC3 does not fall in the supported decidable fragment of first-order logic. In contrast, DistAI never generates an undecidable formula.

Although both DistAI and I4 fail to solve Paxos, a complex and realistic consensus protocol, the reasons for the failures are different. I4 fails because its model checker is unable to produce any candidate invariants. Model checking is complex and quite resource intensive, and I4's authors report its model checker runs out of memory trying to solve Paxos [18]. In contrast, DistAI produces candidate invariants, but it fails because it does not support invariants with existential quantifers, which Paxos requires; I4 also has this limitation. Upon failed refinement, DistAI keeps increasing the formula size until it times out or exhausts memory. By manual

Distributed	Domains		Variables		Refinements		DistAI time(s)	I4 time(s)		FOL-IC3 time(s)	
Protocol		Relations		Literals		Invariants		final	total	∀	default
asynchronous lock server [5]	2	5	3	2	0	12	1.1	generalize fail^{s}		6.9	-*
chord ring maintenance [24]	1	8	3	4	48	163	52.8	586.1‡	594.4	-*	-*
database chain replication [24]	4	13	7	4	158	66	58.8	20.2‡	63.1	-*	Z3 fail
decentralized lock [9]	2	2	4	2	150	16	9.4	generalize fail^{s}		37.1^{d}	Z3 fail
distributed lock [24]	2	4	4	3	82	45	12.6	152.1‡	204.7	1451.3	Z3 fail
hashed sharding [11]	3	3	5	2	0	15	1.1	nondet fail^{s}		9.2	-*
leader election [24]	2	3	6	3	0	17	1.9	4.9‡	4.9	26.3	-*
learning switch [24]	2	4	4	3	8	71	27.6	10.5‡	12.4	-*	-*
lock server [24]	2	2	2	2	0	1	0.8	0.5‡	0.8	0.5	2.1
Paxos [13, 15, 23]	4	9	-	-	-	-	-*	-*	-*	-*	-*
permissioned blockchain [17]	4	10	6	3	2	13	4.9	blackbox fail^{s}		21.2	-*
Ricart-Agrawala [26]	1	3	2	2	0	6	0.9	0.8	0.8	0.7	3.2
simple consensus [11]	3	8	5	3	19	50	23.3	41.8	68.7	-*	-*
two-phase commit [18]	1	7	2	3	3	30	1.9	3.1‡	8.0	3.4	7.9

* Time out after 1 week.
‡ I4 runtimes on our machine are similar (6 out of 7 protocols slightly faster) to those previously reported for I4 [18].
^{s} "generalize fail" means I4's implementation fails to convert invariants from the AVR model checker to generalized universally quantified invariants. "nondet fail" means failed on nondeterministic initialization. "blackbox fail" means error triggered on reasoning of blackbox functions.
^{d} FOL-IC3 initially completed in less than a second, but this turned out to be incorrect due to a bug in the mypyvy protocol specification used by FOL-IC3, which does not exist in the Ivy protocol specification used by DistAI and I4.

Table 1: Evaluation results on 14 distributed protocols from multiple sources.

inspection, we find that DistAI infers all ∃-free invariants for Paxos. FOL-IC3 supports finding invariants with existential quantifiers, but it also fails to solve Paxos, the one protocol in our evaluation with existential quantifiers.

The most common reason overall why I4 fails to solve protocols is its dependency on modeling checking a small size implementation of the protocol to generate candidate invariants. I4 also fails to infer the correct invariants for decentralized lock and asynchronous lock server because it cannot generalize the candidate invariants generated by the model checker for a small size implementation to universally quantified invariants. Although I4 succeeds on lock server, it fails on asynchronous lock server because the latter explicitly models packet loss in the network, resulting in more complex invariants.

DistAI takes a fundamentally different approach that does not require model checking a finite instance. DistAI operates in formula space, allowing it to enumerate invariants that hold for any instance size. It optimizes the enumeration by running protocol simulations across different size systems, but does not rely on the simulations to find candidate invariants, only to reduce the number of invariants it needs to enumerate. By taking this data-driven approach, it is able to produce better initial invariants to achieve greater success with more protocols and guarantee success if there are no invariants with existential quantifiers. Unlike I4, DistAI is simple and self-contained, avoiding the need for, and dependence on, a complex external model checker that, like all complex software, may have bugs.

Permissioned blockchain is another example that demonstrates the effectiveness of DistAI. It has a blackbox Byzantine broadcast protocol as a subprocedure. In permissioned blockchain, n users have a synchronized clock. At epoch E, only one user n_E, the round-robin leader of the epoch, can (optionally) propose a block, if it has found a valid one extending its longest chain. The epoch leader uses the Byzantine broadcast protocol to broadcast the block in the P2P network. An honest user always adds all outstanding transactions in the block it proposed and follows the Byzantine broadcast protocol, while an adversary can neglect certain transactions, delay block proposal, and send conflicting block messages to any node at any epoch, regardless of who the leader is. A Byzantine broadcast protocol satisfies *agreement*, if all honest users always share the same eventual result regardless of the leader is honest or not. A Byzantine broadcast protocol satisfies *validity*, if when the leader is honest, all honest users will eventually decide on the message of the leader. A blockchain satisfies *consistency*, if at any epoch, all honest users have the same view of the blockchain (i.e., no forks or orphaned blocks). That is, for any two honest users at any time, a block is either confirmed by both, or confirmed by none. A blockchain satisfies *liveness*, if all transaction will be confirmed within a finite number of epochs.

DistAI successfully proves that for any Byzantine broadcast procedure that satisfies agreement and validity, the resulting permissioned blockchain satisfies consistency and liveness[1]. The Byzantine broadcast procedure is described by pre-conditions and post-conditions in IVy without the need for an executable implementation. When simulating the blackbox Byzantine procedure, DistAI simply picks a random state that satisfies the post-condition as the execution result. This random selection may leave corner cases uncovered, but the eventual correctness is guarded by the SMT solver, and we monoton-

[1]We prove a variant of the liveness property — if the leader of epoch T is honest, then all transactions before T will be confirmed at T. The original liveness property cannot be encoded as a safety property, thus falling out of the scope of DistAI, I4, and FOL-IC3.

Distributed Protocol	Sample	Enumerate	Refine	Total
asynchronous lock server	0.6	0.0	0.5	1.1
chord ring maintenance	11.1	1.2	40.5	52.8
database chain replication	36.3	0.1	24.4	58.8
decentralized lock	2.1	0.1	7.2	9.4
distributed lock	0.8	0.1	11.7	12.6
hashed sharding	0.6	0.1	0.4	1.1
leader election	0.8	0.1	1.0	1.9
learning switch	19.4	0.3	7.9	27.6
lock server	0.5	0.0	0.3	0.8
permissioned blockchain	3.5	0.1	1.3	4.9
Ricart-Agrawala	0.5	0.0	0.4	0.9
simple consensus	18.8	0.4	4.1	23.3
two-phase commit	0.5	0.1	1.3	1.9

Table 2: DistAI runtime breakdown in seconds for each protocol.

ically weaken the invariant set to reach a correct solution. In contrast, the use of a blackbox procedure poses difficulty and triggers errors in I4. We should note DistAI solves permissioned blockchain for not one, but any valid implementation of the Byzantine broadcast protocol because it does not depend on or require its implementation, a key benefit of our approach.

DistAI also outperforms both I4 and FOL-IC3 in terms of the time required to infer the correct invariants. For I4, we report both the runtime for the final instance size on which the correct invariant is generated, as reported in [18], as well as the total runtime, which includes trying increasingly larger instance sizes that fail until the final instance size succeeds. Except for learning switch, DistAI is about the same or faster than I4 for all of the protocols solved by I4, up to an order of magnitude faster. The runtime comparisons between DistAI and I4 are conservative as they do not include the time required for concretization [18], a step required by I4 to manually introduce additional constraints to the protocol to limit the search space of the model checker. DistAI is also faster than FOL-IC3 for all but the two simplest protocols solved by FOL-IC3, in many cases by more than one to two orders of magnitude. This is because SMT queries are expensive and FOL-IC3 uses them extensively.

Table 1 also shows for DistAI the number of invariants identified for the correct invariant set, the total number of refinement steps required (i.e., the number of times Algorithm 3 is called), the total number of quantified variables of all domains in the final template used, and the maximum number of literals used for each protocol. Most protocols have a maximum number of literals of 2 or 3, and in two cases 4. This validates our key assumption that inductive invariants of distributed protocols should be human-readable and concise. DistAI uses invariant refinement to address missing cases during sampling for all but the five simplest protocols, Ricart-Agrawala, hashed sharding, leader election, lock server, and asynchronous lock server, in which no refinement is needed as the subsample set is complete and the correct invariant is learned without refinement.

Runtime breakdown. Table 2 provides a breakdown of the total runtime using DistAI for each protocol. One can see the bottleneck is either sampling or refinement, but not enumeration. Sampling is expensive when the argument space for actions is sparse because DistAI randomly selects arguments so it can end up trying many arguments that are invalid for each set of valid arguments, increasing the simulation runtime. This is the reason why sampling is most expensive for database chain replication, a protocol that guarantees serializability and atomicity for distributed databases. A transaction is split into subtransactions that operate sequentially on data that is sharded across multiple nodes. For one subtransaction to commit, it must operate on the correct node and satisfy a set of constraints (e.g., no uncommited previous writes). Most randomly selected subtransactions will not satisfy these constraints. As a result, sampling spends significant time finding valid arguments because most arguments are invalid.

Sampling is also expensive for learning switch, the only protocol for which DistAI is slower than I4. One reason is the argument space for actions is sparse, so it takes a while to find a data sample, but the other is because the subsample space is too large to explore. With learning switch, each node maintains a routing table that matches destination addresses to outbound ports (i.e., neighbors), and updates the table upon receiving new packets. It has a 3-ary single-domain relation $route\_tc(N_1,N_2,N_3)$, which means the routing trace from N_2 to N_1 includes N_3. Under template $\forall N_1\ N_2\ N_3$, this single relation yields 27 predicates $(route\_tc(N_1,N_1,N_1), route(N_1,N_1,N_2),...)$. There are 60 predicates across all relations, meaning that each subsample is a 60-bit vector, so the candidate subsample space has size 2^{60}. Although valid subsamples are sparse, DistAI generates 33K subsamples before it cannot find anymore and terminates. This takes a while.

Refinement can be the dominant factor in performance, as is the case for chord ring maintenance, database chain replication, and distributed lock, but Table 2 shows that DistAI is successful overall at avoiding substantial SMT query costs as refinement runtime, which includes the cost of IVy checking the initial candidate invariants, is modest in most cases.

Figure 5 shows how sampling helps reduce the cost of refinement for the simple consensus protocol. DistAI has provable guarantees to find the correct invariants for any sample sizes, but if the number of samples is too small (e.g., 100 in Figure 5), it takes much longer due to many more SMT queries on refinement. Increasing the number of samples increases sampling time roughly linearly but decreases refinement time roughly exponentially. However, once a minimum threshold of samples is met, it becomes more of an even tradeoff. Sampling can be faster by obtaining fewer samples, but because more corner cases are missing, the refinement process takes longer to "fix" the invariants through monotonic weakening. Conversely, more samples require more time to simulate the protocol, while the refinement process will be faster. The default sampling parameters, discussed in Section 3 and used for all experiments, resulted in 50K samples for the simple consensus protocol.

We also reran the protocol experiments with DistAI for other

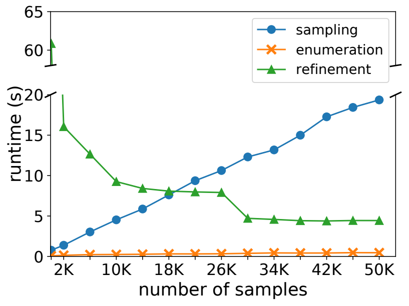


Figure 5: Runtime breakdown of DistAI on simple consensus.

sampling parameters, ranging from $MA = 25$ to $MA = 100$, and $SD = 2$ to $SD = 5$. In all cases, DistAI was able to solve the same 13 protocols with mostly similar runtimes and in the worst case, three times slower runtimes than the defaults. Detailed runtimes are omitted due to space constraints.

7 Related Work

Much work [10, 14, 16, 28, 31–33] has shown how to verify the correctness of distributed protocols and distributed systems given inductive invariants, but they rely on a user or external system to provide them. Various approaches have explored learning invariants for distributed protocols. Dinv [8] identifies and tracks critical variables in distributed systems, and then infers likely correct invariants over these variables with Daikon [2], a data-driven invariant learning tool. The invariants inferred using Dinv are not guaranteed to be valid and may not be inductive. Phase-PDR$^{\forall}$ [5] showed how to generate invariants for distributed protocols if users can provide phase structures. In contrast, DistAI is fully automated and does not require users to provide any additional knowledge about the protocol.

More recently, approaches have been developed for learning inductive invariants for distributed protocols. I4 [18] is the first. Its key idea is to use model checking on a finite protocol instance to generate candidate invariants. Although generated invariants by model checking some small instance always generalized in [18], this is not guaranteed. Our evaluation shows several protocols for which generalizing fails. I4 does not support existential quantifiers and also requires a manual concretization step. In contrast, DistAI is fully automated and provably guaranteed to learn inductive invariants without existential quantifiers. FOL-IC3 [11] can learn invariants with existential quantifiers by invoking an SMT solver to generate a candidate formula that can separate a positive and a negative example set. However, its heavy use of an SMT solvers slows its performance to the point that in practice, it fails to find inductive invariants for protocols that are efficiently handled by DistAI.

SWISS [9] is concurrent work that searches for invariants by template enumeration and checking candidate invariants with SMT queries. It does not do sampling, enumerating strongest possible invariants, or monotonic refinement, but does incorporate existential quantifiers in its invariant templates and uses counterexamples to prune the formula search space. In its reported evaluation, SWISS finds a correct existentially quantified invariant for Paxos, but fails or takes orders of magnitude more time than DistAI to find correct invariants for many other protocols listed in Table 1, despite being multithreaded.

Many automated invariant inference tools have been built for systems verification. Most of these tools focus on finding invariants in sequential programs with loops. Traditional methods use symbolic reasoning to infer invariants [6, 12], while recently data-driven methods using execution traces and/or counterexamples have shown promise. Guess-and-check, Numinv, and G-CLN recast invariant inference as a curve-fitting task on execution traces, and learn loop invariants represented by polynomials of program variables [20, 27, 29, 34]. ICE-DT and LoopInvGen (PIE) apply decision tree learning and PAC learning on counterexamples and iteratively refine the invariants [7, 21, 22]. FreqHorn exploits both syntax and data in its inference tool and learns $\forall$-quantified array invariants [3, 4]. Recently, data-driven invariant inference has been used in other domains, such as solving CHC clauses [35] and proving properties on inductive algebraic data types [19]. None of these methods consider nondeterminism in concurrent or distributed settings, thus they cannot be directly applied to distributed protocols.

8 Conclusions

DistAI is a fully automated, data-driven methodology for learning inductive invariants for distributed protocols. DistAI uses data samples from protocol simulation to enumerate the strongest possible set of candidate invariants, then feeds them to an SMT solver to check if adding them to the safety property is inductive. If any invariants fail, DistAI refines them by monotonically weakening the invariant set and tries again with the solver until it eventually succeeds. Starting with small invariant formulas and strongest possible invariants based on data from protocol simulation avoids large and frequent SMT queries, improving performance. Starting with strongest possible invariants makes refinement via monotonic weakening possible, enabling DistAI to provably guarantee that it will learn the correct inductive invariant set without existential quantifiers in finite time. Our evaluation shows that DistAI successfully learns inductive invariants for real distributed protocols and outperforms alternative methods, solving almost 50% more protocols and doing so up to one to two orders of magnitude faster.

Acknowledgments

Manos Kapritsos provided helpful comments on earlier paper drafts. This work was supported in part by an Amazon Research Award, a Guggenheim Fellowship, an NDSEG Fellowship, DARPA contract N6600121C4018, an NSF CAREER award, and NSF grants CNS-2052947 and CCF-1918400.

References

[1] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, pages 193–206, August 2015.

[2] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, December 2007.

[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] Yotam MY Feldman, James R Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV '19)*, pages 405–425, July 2019.

[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] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, pages 1149–1159, May 2018.

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

[10] 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.

[11] 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.

[12] 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.

[13] LESLIE LAMPORT. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

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

[15] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[16] 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)*, page 357–370, January 2016.

[17] Kuan-Ching Li, Xiaofeng Chen, Hai Jiang, and Elisa Bertino. *Essentials of Blockchain Technology*. CRC Press, 2019.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Saswat Padhi, Rahul Sharma, and Todd Millstein. Loopinvgen: A loop invariant generator based on precondition inference. *arXiv preprint arXiv:1707.02029v4*, October 2019.

[23] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: Decidable reasoning about distributed protocols. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, volume 1, pages 1–31, October 2017.

[24] 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.

[25] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNET: A greybox fuzzer for network protocols. In *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification (ICST '20)*, pages 460–465, October 2020.

[26] Glenn Ricart and Ashok K Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[27] 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.

[28] Ilya Sergey, James R Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. In *Proceedings of the ACM on Programming Languages (POPL)*, volume 2, pages 1–30, December 2018.

[29] 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.

[30] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D. Keromytis, and Suman Jana. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P '17)*, pages 521–538, May 2017.

[31] 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)*, page 662–677, June 2018.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

A Artifact Appendix

Abstract

An accompanying artifact includes all DistAI source code as well a docker image. Instructions are provided to reproduce the results in Table 1, Table 2, and Figure 5. The artifact can also be used to learn inductive invariants for other distributed protocols written in IVy.

Scope

The docker image can reproduce Table 1, Table 2, and Figure 5. The file `https://github.com/VeriGu/DistAI/blob/master/docker_usage.md` provides instructions to set up and use the docker. Alternatively, one can build DistAI from source, and reproduce the DistAI results in Table 1, Table 2, and Figure 5. The README file (https://github.com/VeriGu/DistAI/blob/master/README.md) describes how to use DistAI to learn inductive invariants for other distributed protocols written in IVy.

Contents

The README file describes the structure of the artifact. The `src-py` and `src-c` directories include the Python portion and C++ portion of the source code. The `benchmarks` directory includes IVy specifications for the 14 protocols used in the evaluation.

Hosting

The artifact is hosted on GitHub in the repository https://github.com/VeriGu/DistAI. Future updates will be pushed to the master branch, and we encourage you to use the latest version available.

Requirements

The docker image has all dependencies installed. The installation guide (https://github.com/VeriGu/DistAI/blob/master/install.md) provides instructions to build DistAI from source. Note that IVy only works on Python 2, while the source code of DistAI is written in Python 3 and C++. The artifact has been tested on Ubuntu 20.04.1 LTS with ms-ivy 1.7.0, Python 2.7.18, and Python 3.8.5.