

# PINFER: Automatically Learning Specifications for Distributed Systems from Event Traces

Mike He<sup>†</sup> Ankush Desai<sup>\*</sup> Aishwarya Jagarapu<sup>\*</sup> Doug Terry  
Sharad Malik<sup>†</sup> Aarti Gupta<sup>†</sup>

<sup>†</sup>Princeton University <sup>\*</sup>Amazon Web Services

## Abstract

Reasoning about the correctness of distributed systems is a significant challenge, with precise correctness specifications serving as an essential prerequisite. However, identifying and formulating these specifications remains a major hurdle for system developers in practice. PINFER addresses this challenge by automatically learning these specifications from *observable event traces* of messages exchanged in distributed systems. PINFER uses a specialized grammar for the target specifications based on events, including support for forall-exists ( $\forall\exists$ ) quantifiers over events that is essential for capturing specifications of complex protocols. It uses a novel learning procedure where a grammar-based enumerative search provides effective control over the scope of dynamic learning from event traces. We evaluated PINFER on well-known distributed protocols as well as real-world industrial case studies. Our results demonstrate that for these benchmarks PINFER successfully learns *all* protocol specifications.

## 1 Introduction

Developers of distributed systems face the daunting challenge of reasoning about system correctness in the presence of myriad interleavings of messages and potential failures. Even before attempting to reason about correctness, a fundamental challenge emerges – the formulation and comprehension of *correctness specifications* themselves. These specifications, expressed as safety or liveness properties, serve as the essential foundation for the validation approaches used across industry or academia, from lightweight testing [15, 46, 60] or state space exploration via model checking [13, 24, 32], to formal proofs using theorem proving [25, 38, 47, 48, 53] or deductive verification [22, 43, 44, 59]. The importance of formal specifications is well-recognized in industry practice [6, 9, 34, 41] as they serve dual purposes: as important artifacts against which system behavior can be validated, and for providing valuable insights into how the system behaves under various inputs and failures.

Existing techniques for testing or verification of distributed systems predominantly operate under the assumption that specifications are provided a priori by users, thereby transferring the burden of producing comprehensive specifications to the developers. Through our extensive experience using

the P modeling framework [3, 13] to reason about the correctness of industry-strength distributed services at AnonCompany,<sup>1</sup> we have consistently observed that the formulation of correctness specifications constitutes the most significant and time-consuming challenge encountered by service teams. PINFER mitigates this burden on end users by automatically inferring protocol specifications for distributed systems with minimal user intervention.

We identified two key design principles as the requirements that PINFER satisfies, *to be applicable in practice and scalable for real-world distributed systems*. First **R1**: PINFER learns the *protocol specifications* deemed important by system builders in practice. In other words, the target specifications that PINFER learns are sufficiently expressive to capture most (if not all) protocol specifications for real-world distributed systems. These protocol specifications are typically documented in the original research articles proposing the protocol as key theorems, lemmas, or corollaries; or in industrial contexts, as global and component-level specifications articulated by developers in their design documents (examples are shown in Table 4, §6).

Second, **R2**: PINFER is agnostic to the system implementation and development language, and treats the system and its components (or nodes) as black boxes with externally observable behavior. We consider traces, each of which is a sequence of messages exchanged in a distributed system, as its observable behavior. This allows PINFER to be applied in many settings since the traces can be generated from any stage of the development cycle, e.g., by a model checker used during design analysis, collected from test infrastructure when testing the implementation, or extracted from production service logs. Representing specifications in terms of the messages exchanged allows the learning algorithm to focus on the interfaces between components, i.e., the messages, rather than irrelevant internal details of the nodes.

### 1.1 PINFER: Key Ideas

There are two key insights in our approach that enable PINFER to satisfy these requirements.

**(1) Protocol specifications over messages.** The first insight is based on the observation that most protocol specifications for distributed systems can be formulated over messages using (restricted) first-order logic (FOL). These

<sup>1</sup>company name elided for anonymity

| Specification Description                    | Specification Formula   |
|--|---|
| <i>External Consistency</i> (Spanner [11])   | $\forall e_0, e_1 : eCommit. [e_0.commit < e_1.start] \rightarrow e_0.ts < e_1.ts$                                      |
| <i>Election Safety</i> (Raft [42])           | $\forall e_0, e_1 : eElected. [e_0.term = e_1.term] \rightarrow e_0.leader = e_1.leader$                                |
| <i>Update Prop.</i> (Chain Replication [51]) | $\forall e_0, e_1 : eNodeLog. [e_0.pos \leq e_1.pos] \rightarrow e_1.log \preceq_u e_0.log$                             |
| <i>Whitelist Safety</i> (Firewall [40])      | $\forall e_0 : eRecv. [e_0.allowed = \top] \rightarrow \exists e_1 : eGrant. [e_1 \prec e_0] \wedge e_0.src = e_1.host$ |
| <i>Quorum Votes</i> (Consensus [39])         | $\forall e_0 : eDecide. \exists_{\geq quorum} e_1 : eVote. [e_1 \prec e_0] \wedge e_0.ballot = e_1.vote$                |

**Table 1. Example specifications expressed as formulas over events**, with  $\boxed{G}$ ,  $\boxed{W}$ ,  $\boxed{H}$

formulations employ straightforward predicates including equality, numerical comparisons, and temporal operations such as happens-before relationships [31]– to define relationships between messages. We refer to a message exchanged in a distributed system as an *event*, which has an accompanying data *payload*. Table 1 shows example specifications of well-known protocols expressed over events and their payload fields. For example, the *External Consistency* of Google Spanner [11] says that if a transaction commits before another transaction starts, then its commit timestamp is less than the commit timestamp of the other transaction. This is expressed as a formula in terms of two *eCommit* events ( $e_0$  and  $e_1$ ) that are sent when a transaction commits, where the if-then form is expressed by an implication ( $\rightarrow$ ) between two subformulas. Note that the subformulas use standard arithmetic operators ( $<$ ) over the fields of the event payloads, and the quantifiers ( $\forall$ ) on events indicate that this formula must hold over all possible pairs of events  $e_0$  and  $e_1$  in a trace of the system. The other specifications are described in detail later, §3.1. While explicitly focusing on events, in cases where exposing some internal state is required, a user can make minimal relevant state *visible* in event payloads.

**(2) Structure of specifications.** The second insight is that a protocol specification over events can be decomposed into the following parts (as marked in Table 1): events over which the specification is defined, *Guards*  $G$  capturing *control conditions*, *Witness*  $W$  (optionally) capturing the *existence* of certain events and *Hypotheses*  $H$  stating conditions that must hold under the Guards and the Witnesses. For instance, the *Whitelist Safety* specification of a Firewall protocol [40] in Table 1 is defined over *eRecv* (receiving) and *eGrant* (granting) events. The Guard  $G$  states the control condition of allowing to receive the event. The Witness  $W$ ,  $e_1 \prec e_0$  with the traditional happens-before [31] relation, states that when receiving is allowed, then there must *exist* some granting events happened before. Finally, the Hypotheses  $H$  states that an earlier granting events must have granted the permission to the sender. This structure enables PINFER to effectively learn specifications through a novel combination of a static enumerative search (over events involved in the specifications,  $G$  and  $W$ ) and dynamic learning (to learn  $H$ ).]

## 1.2 Results and Contributions

We have implemented PINFER as a part of the P modeling framework [3, 13], leveraging Daikon [14] for dynamic learning. We constructed a suite of benchmarks with 11 well-known protocols and 3 proprietary industrial case studies written in P. We use 43 known safety specifications of these protocols as a challenge set of protocol specifications for evaluating PINFER, including specifications express Linearizability [23] and Snapshot Isolation [7]. Our evaluations (§6) show that PINFER can automatically learn a 28 of 43 protocol specifications from the challenge set *without* any user guidance or hints. Furthermore, with light-weight user guidance (discussed in detail in §6), PINFER learns the remaining 15 protocol specifications. Some specifications in our challenge set involve *both* temporal relations between events *and* relationships between event payloads, and require forall-exists ( $\forall\exists$ ) quantifiers.

**Contributions.** We summarize our contributions as follows:

- (1). We present PINFER, the first largely automated specification learning framework that learns protocol specifications with both temporal relations between events, predicates over event payload, and the use of  $\forall\exists$  quantifiers.
- (2). The learning procedure in PINFER is a novel combination of static enumerative search that allows fine-grained control on the scope of dynamic learning on event traces.
- (3). We have implemented PINFER as an extension to the P framework [13], and report evaluation results on a suite of benchmarks that demonstrate its effectiveness in learning complex specifications for many well-known and some proprietary benchmarks.

## 2 Overview of PINFER Framework

In this section, we present the formula template the workflow for learning specifications. Then, we introduce the P language using a running example, and use an example to illustrate an overview of the main steps in PINFER.

### 2.1 Formula Template and Workflow

PINFER learns specifications using the following *formula template* that is *sufficiently expressive* for capturing many safety and liveness properties in protocol specifications:

$$\phi : (\forall \vec{e}_i)^+. \boxed{G(\vec{e}_i)} \rightarrow (\exists \vec{e}_j)^*. \boxed{W(\vec{e}_i, \vec{e}_j)} \wedge \boxed{H(\vec{e}_i, \vec{e}_j)} \quad (1)$$

It uses universal quantifiers ( $\forall$ , at least one) as well as existential quantifiers ( $\exists$ , optionally) over vectors of events ( $\vec{e}_i, \vec{e}_j$ , respectively). While this allows capturing relationships between any fixed number of event types in the traces, we found that it is enough in practice to learn specifications that are quantified over just *two types of events*. The formula template uses the if-then form that occurs commonly in specifications, to capture conditions under which certain relationships hold on events. It further differentiates the subformulas shown as Guard ( $\boxed{G}$ ), Witness ( $\boxed{W}$ ), and Hypothesis ( $\boxed{H}$ ). Note that although this formula template resembles a popular fragment of FOL called EPR [?], PINFER does not rely on decidability of the associated logic or decidable models for learning specifications, unlike many verifier-based approaches [20, 28, 39, 44, 57] (detailed discussion in related work, §7.)

In general, learning specifications poses several challenges, since there is an inherent trade-off between the expressiveness of the learning targets and search efficiency. PINFER carefully constrains the search space for learning, and provides an iterative approach to increase the complexity of target specifications. Furthermore, the learning procedure in PINFER is fully automated, with the ability to leverage user guidance if required. Specifically, PINFER uses a static enumerative search (over  $\vec{e}_i, \vec{e}_j, G, W$ ) to provide *critical filters that control the scope of dynamic learning* (for  $H$ ), where the formula template (Eqn 1) provides a structured way to decompose the overall search. PINFER first *selects* the events  $\vec{e}_i, \vec{e}_j$  based on causality-related heuristics – this is most critical in constraining the search space to focus on specific events of interest. Then PINFER *selects* candidates for Guards  $G$  and Witnesses  $W$  within an iterative search – these identify important control conditions that *must occur*, such as in different phases of a protocol, or in certain witnesses that occur later. For each enumerated candidate, PINFER invokes a dynamic learner (such as Daikon [14]) to *learn* the Hypotheses  $H$  from event traces. These represent relationships that hold on the events/payloads under the control conditions ( $G, W$ ), and which can be learned automatically by a dynamic learner. Additionally, PINFER uses logic-based pruning procedures in post-processing to soundly reduce the number of specifications learned after dynamic learning.

## 2.2 Background: P Modeling Language

P [3, 13] is a state-machine-based programming language designed for formal modeling and analysis of distributed systems. It enables developers to model system designs as collections of communicating state machines, facilitating checking of both safety and liveness properties. For instance, Amazon Web Services (AWS) used P to validate the strong consistency protocol in Amazon S3 [9], gaining confidence in its correctness during the transition from eventual to strong consistency. Similarly, DeepSeek employed P to check

```

1  type tNodeId = int;
2  event eNominate: (vote: tNodeId);
3  event eElectedAsLeader: (nodeId: tNodeId);
4
5  machine Node {
6    var id: tNodeId; # unique identifier of the node
7    var right: machine; # the node to its right
8    start state Init {
9      entry (cfg: (nodeId:tNodeId, next:machine)) {
10         id = cfg.nodeId;
11         right = next;
12         goto Nominating;
13       }
14     }
15     state Nominating {
16       entry {
17         send right, eNominate, (vote=id,);
18       }
19       on eNominate do (payload:(ballot:tNodeId)) {
20         if (payload.vote == id) {
21           announce eElectedAsLeader, (nodeId=id,);
22           goto Won;
23         } else if (payload.vote > id) {
24           send right, eNominate, (vote=payload.vote,);
25         } else {
26           send right, eNominate, (vote=id,);
27         }
28       }
29     }
30     state Won { ignore eNominate; }
31 }

```

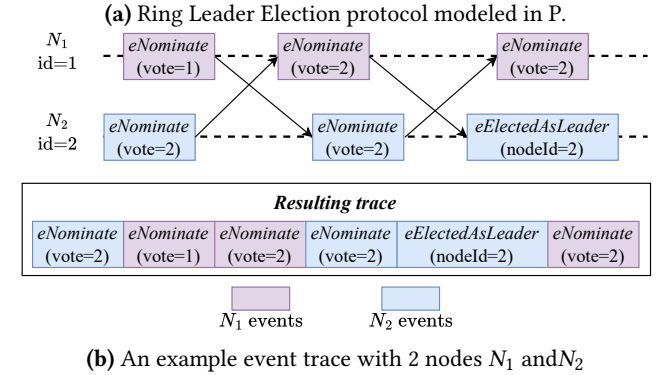


Figure 1. Ring Leader Election protocol in P

the correctness of the Fire-Flyer File System (3FS), a high-performance distributed file system [5], showcasing its effectiveness in analyzing complex distributed systems across diverse applications. In P, developers specify the desired correctness specifications as invariants over the events exchanged by the state machines, and use the PChecker to check that the model satisfies the desired specifications using random state space exploration.

## 2.3 Example: Ring Leader Election protocol

A P program comprises state machines communicating asynchronously with each other using events accompanied by typed data values. Each machine has an input buffer, event handlers that are executed on receiving an event, and a local store. The machines run concurrently, receiving and sending events, creating new machines, and updating the local store.

We use the well-known Ring Leader Election protocol [10] as our running example. In this protocol, nodes are arranged in a ring topology and assigned a unique, totally ordered identifier (id). To initiate the leader election, each node sends its id to the node on its right. Upon receiving an id, a node compares it with its own id. If they are equal, the node declares itself the leader. Otherwise, it forwards the greater id to the node on its right. Ultimately, the node with the highest id elects itself as the *unique* leader.

Figure 1a shows the state machine Node (line 5) in the Ring Leader Election protocol. The event declarations (lines 2, 3) specify the names and payload types of the events used in the Ring Leader Election protocol. For example, the *eNominate* event (line 2) has a payload with a vote field of type *tNodeId*.

The state machines in P have local variables and states. Each state may have an entry function that is executed on entering the state through a transition. After executing the entry function, the machine tries dequeuing an event from its buffer and executes the associated event handler. For instance, in *Nominating* state, the machine has a handler for *eNominate* event (line 19), where the argument of the handler takes the value of the event payload. Machines may send an event to another machine or broadcast an event, by using *send* (line 17) or *announce* (line 21), respectively. Executing *goto* statements (e.g., line 12) transitions the machine to another state. When a Node state machine is created, it receives a configuration with its assigned id, and right (the node to its right). After initialization, the machine transitions to *Nominating* state and sends its id to right using the *eNominate* event. Upon receiving an *eNominate* event, it compares vote to its own id. If equal, then it announces an *eElectedAsLeader* event with its id and transitions to the *Won* state. Otherwise, it sends an *eNominate* event to right with the greater id as payload.

Figure 1b shows an example execution and the resulting event trace generated from an instance of the P model with two nodes  $N_1, N_2$ . Along the dashed lines, events are ordered by the time of occurrence. The arrows show causal relations between events. The event trace is a sequence of events, ordered by the time of occurrence in the observed execution.

## 2.4 Specifications of Ring Leader Election protocol

The safety specification of the protocol is that a *unique* leader with the *highest* id is *elected*, which can be encoded with the following formulas over the *eElectedAsLeader* ( $\mathcal{E}$ ) and *eNominate* ( $\mathcal{N}$ ) events:

$$\forall e_0, e_1 : \mathcal{E}. e_0.\text{nodeId} = e_1.\text{nodeId} \quad (2)$$

$$\forall e_0 : \mathcal{E}, e_1 : \mathcal{N}. e_0.\text{nodeId} \geq e_1.\text{vote} \quad (3)$$

$$\forall e_0 : \mathcal{E}. \exists e_1 : \mathcal{N}. e_1 \prec e_0 \wedge e_0.\text{nodeId} = e_1.\text{vote} \quad (4)$$

Eqn (2) states that the value of the *nodeId* field in  $\mathcal{E}$  events is *unique*, which implies the uniqueness of the leader under the setting where each node gets a unique id. Eqn (3) states

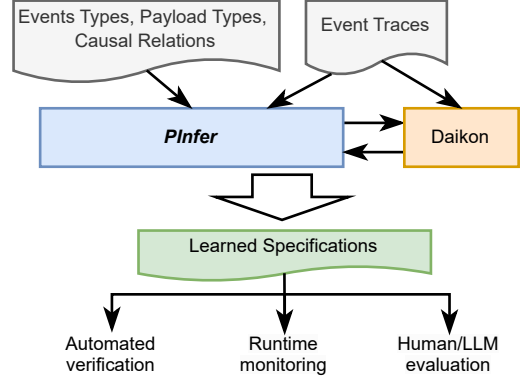


Figure 2. Workflow of PINFER

| Events  | $e_0 : \mathcal{E}, e_1 : \mathcal{N}$  |
|---|---|
| <b>Generated Predicates <math>\mathcal{P}</math></b>  | $e_0 \prec e_1, \quad e_1 \prec e_0$<br>$e_0.\text{nodeId} = e_1.\text{vote}$<br>$e_0.\text{nodeId} < e_1.\text{vote}$<br>$e_0.\text{nodeId} > e_1.\text{vote}$<br>$e_0.\text{nodeId} \leq e_1.\text{vote}$<br>$e_0.\text{nodeId} \geq e_1.\text{vote}$ |
| <b>Event Combination (a)</b>  | $\forall e_0 \forall e_1$   |
| Enumerated Guard $G$  | $\top, e_1 \prec e_0, \dots$  |
| Learned Hypothesis $H$  | $e_0.\text{nodeId} \geq e_1.\text{vote}$  |
| <b>Event Combination (b)</b>  | $\forall e_0 \exists e_1$   |
| Enumerated Guard $G$  | $\top, \dots$   |
| Enumerated Witness $W$  | $e_1 \prec e_0, \dots$  |
| Learned Hypothesis $H$  | $e_0.\text{nodeId} = e_1.\text{vote}$   |
| <b>Learned specifications (before pruning)</b>  |   |
| (a1) $\forall e_0 : \mathcal{E}. \forall e_1 : \mathcal{N}. e_0.\text{nodeId} \geq e_1.\text{vote}$                           |   |
| (a2) $\forall e_0 : \mathcal{E}. \forall e_1 : \mathcal{N}. e_1 \prec e_0 \rightarrow e_0.\text{nodeId} \geq e_1.\text{vote}$ |   |
| (b) $\forall e_0 : \mathcal{E}. \exists e_1 : \mathcal{N}. e_1 \prec e_0 \wedge e_0.\text{nodeId} = e_1.\text{vote}$          |   |

Table 2. Example illustrating main steps of PINFER

that the value of *nodeId* in an  $\mathcal{E}$  event is the *highest* among the votes in  $\mathcal{N}$  events. Finally, Eqn (4) states that for every  $\mathcal{E}$  event, the *nodeId* must be voted in some  $\mathcal{N}$  events that *happened before* (denoted as  $e_1 \prec e_0$ ) [31] the  $\mathcal{E}$  event. When interpreted over a single trace, an event  $e_0$  happened before another event  $e_1$  if  $e_0$  appears earlier in the trace than  $e_1$ . For instance, in the resulting trace in Figure 1b, the  $\mathcal{N}$  event with *vote*=1 (second box) appears earlier than the  $\mathcal{E}$  event (fifth box) and therefore happened before the  $\mathcal{E}$  event.

## 2.5 From Event Traces to Specifications

The top-level workflow of PINFER is shown in Figure 2. PINFER needs two kinds of inputs: (1) a set of traces (which could be generated from a P program), and (2) information about events in the trace: event types, their payload types, and causal relations between events. In the PINFER prototype, we perform a simple static analysis over the P program to extract the latter. The outputs of PINFER are learned specifications



that can be reviewed by users and used in downstream tasks for verification or runtime monitoring.

We will use the Ring Leader Election protocol to illustrate the important steps in the PINFER framework (along with section references that describe the technical details).

**Inputs to PINFER.** For our example, suppose event traces are generated by using PChecker [3] on the P program shown earlier. PINFER uses a simple static analysis on the P program to determine information about events. It identifies two types of events,  $eNominate(N)$  and  $eElectedAsLeader(E)$ , where the payloads have fields `vote` and `nodeId`, respectively, of type `tNodeId`, as declared in lines 2 and 3 in Figure 1a.

**Determining Event Combinations (§4.1).** As a first step, PINFER uses heuristics based on causal relations to determine events and their respective quantifiers for learning specifications— we call these *event combinations*. In our example, PINFER identifies the following pairs of events to be of interest:

Pair 1: an  $E$  event and an  $N$  event

Pair 2: two  $E$  events

Pair 3: two  $N$  events. We will use Pair 1 as an illustration, with the results of intermediate steps listed in Table 2; the steps are similar for the other event pairs.

**Generating Predicates  $\mathcal{P}$  (§4.2).** Given the events and their types, shown in the top row of Table 2, PINFER generates a set of predicates  $\mathcal{P}$  shown in the next row, each of which uses standard operations over the events and their payloads.

**Learning Procedure (§5).** Then, PINFER performs the learning procedure, which employs a novel combination of enumerative search (§5.1) and dynamic learning (§5.2). The enumerative search enumerates a *guard*  $G$  and (optionally) a *witness*  $W$  (when targeting formulas with existential quantifiers) from the set of predicates  $\mathcal{P}$ . With the enumerated  $G$  (and an optionally enumerated  $W$ ), PINFER then invokes a dynamic learner (Daikon [14]) that learns *hypotheses*  $H$ . Some examples of enumerated  $G$ ,  $W$ , and the corresponding  $H$  learned for them, are shown for the Event Combinations labeled (a) and (b) in the table, along with the corresponding learned specifications. Note that the learned specification (b) from the Event Combination (b) is the same as Eqn (4) shown earlier (§2.4). This is learned because in all traces, the value of `nodeId` in an  $E$  event has appeared as a vote from some  $N$  event that occurs before the  $E$  event.

**Logic-based pruning procedures (§5.3).** The learning procedure can result in learning several specifications, many of which are redundant (e.g., due to subsumption). PINFER applies logic-based pruning procedures to eliminate specifications due to tautology and subsumption, resulting in a much smaller set to report to the user. In our example, the learned  $H$  in Event Combination (a) is also learned by Daikon when the enumerated  $G$  is  $\top$  (since the `nodeId` is higher than the vote in *all*  $eNominate$  events). The pruning procedure in

|                |   |
|----------------|---|
| Vars           | $v \in V$   |
| Constants      | $c \in C$   |
| Event Vars     | $e \in E$   |
| Event Types    | $\tau$  |
| Terms          | $T ::= C \mid E.v \mid f(\vec{T}) \mid \text{indexof}(E)$   |
| Preds          | $P ::= P_E \mid P_F$  |
| Event Preds    | $P_E ::= E \equiv E \mid E \prec E$   |
| Field Preds    | $P_F ::= T = T \mid T \neq T \mid T < T \mid T > T \mid$<br>$T \leq T \mid T \geq T \mid uP(uf(\vec{T}))$ |
| SetCardinality | $SC ::= \geq 1 \mid = C \mid \leq C \mid \geq C$  |

|  |
|--|
| $\phi_{\forall} ::= (\forall e_i : \tau_i)^+ . \boxed{\bigwedge \vec{P}} \rightarrow \bigwedge \vec{P}$  |
| $\phi_{\forall\exists} ::= (\forall e_i : \tau_i)^+ . \boxed{\bigwedge \vec{P}} \rightarrow (\exists_{SC} e_j : \tau_j)^+ . \boxed{\bigwedge \vec{P}} \bigwedge \vec{P}$ |

**Figure 3.** Grammar for Target Formulas

PINFER identifies this and keeps only specification (a1) with  $G = \top$ , resulting in Eqn (3) shown earlier (§2.4).

### 3 Formula Template: Grammar & Examples

In this section, we define the formal grammar and semantics for the formula template used in PINFER, and describe example specifications that illustrate its expressive power.

#### 3.1 Grammar and Semantics

PINFER uses two classes in formula templates, shown as boxed rules in the grammar listed in Figure 3. The first class, shown as  $\phi_{\forall}$ , uses only forall ( $\forall$ ) quantifiers over event variables  $\vec{e}_i$ . The second class, shown as  $\phi_{\forall\exists}$ , uses additional nested existential ( $\exists$ ) quantifiers over event variables  $\vec{e}_j$ . Variables  $\vec{e}_i$  and  $\vec{e}_j$  have types  $\tau_i$  and  $\tau_j$ , respectively. (In the sequel, we will assume that all events are typed-events.) Additionally, the existential quantifiers may be augmented by a Set Cardinality ( $SC$ ) constraint, to be explained shortly.

In the body of the formula templates, each of the boxes  $\left( \boxed{G(\cdot)}, \boxed{W(\cdot)}, \boxed{H(\cdot)} \right)$  is a conjunction of *atomic predicates*  $P$ , which are defined over events or terms. The grammar for predicates (Preds  $P$ ) and terms (Terms  $T$ ) is shown above the box in Figure 3. Notably, this grammar is sufficient to express all protocol specifications in a comprehensive set of benchmarks, as shown later in our evaluations (§6.3).

**Grammar for Terms and Predicates.** As shown in the grammar rules, terms ( $T$ ) are comprised of constants ( $C$ ), fields of event payloads ( $E.v$ ), function applications over terms ( $f(\vec{T})$ ), and applications of `indexof` over event variables (`indexof( $E$ )`). For function applications over terms ( $f(\vec{T})$ ), PINFER supports standard arithmetic functions over integers and floating points, and a `sizeof` function for a

list or set that returns its size. The function  $\text{indexof}(e)$  is interpreted as the index of event  $e$  in a trace.

Predicates ( $P$ ) are of two kinds: (1) Event Preds ( $P_E$ ) that relate event variables, and (2) Field Preds ( $P_F$ ) that relate terms built from payload fields of events. The former includes  $\prec$ , the standard happens-before relation [31] to express temporal relationships between events. The latter includes standard operators over suitably-typed terms: equality ( $=$ ), inequality ( $\neq$ ), numerical comparisons ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ).  $P_F$  also includes user-defined predicates (denoted  $uP$ ) over terms with user-defined function applications over terms (denoted  $uf(\vec{T})$ ). The grammar can be easily extended, supported by an evaluator from concrete values of event fields in traces, to concrete values (for terms) and to true/false (for predicates).

**Support for Existential Quantification.** We support forall-exists ( $\forall\exists$ ) quantifiers in  $\phi_{\forall\exists}$ , since this can capture both safety (e.g., for all incoming messages, there exists a handshake) and some liveness properties (e.g., for all client requests, there exists a server response). We refer to an existential quantifier labeled with an SC constraint as an *augmented existential quantifier* — it allows quantification over a set of witness events, where the cardinality of the witness set satisfies the labeled SC constraint. This is very useful for expressing quorums in consensus protocols (with a detailed example with learning shown in §5.2). The standard semantics of an existential quantifier is equivalent to the SC constraint:  $\geq 1$  (as shown in the rule for SC).

**Semantics of Target Formulas over Traces** We interpret a target formula over a finite trace  $\mathcal{T}$ , which is a sequence of events ordered by the time of occurrence. A formula  $\phi$  is true on a trace  $\mathcal{T}$ , i.e.,  $\mathcal{T} \models \phi$ , if and only if the events in the trace satisfy the standard interpretations of quantifiers, logical operations, equality, arithmetic operations, temporal precedence, etc. For example, the interpretation of temporal precedence  $\prec$  is defined over a trace  $\mathcal{T}$  such that if an event  $e_i$  appears before another event  $e_j$  in the trace, then  $\mathcal{T} \models e_i \prec e_j$ . We extend the semantics of a target formula  $\phi$  to a set of traces  $\mathbb{T}$ , where  $\mathbb{T} \models \phi$  if and only if  $\forall \mathcal{T} \in \mathbb{T}. \mathcal{T} \models \phi$ . We use this semantics for evaluating the truth of a target formula on the given set of traces, without requiring a decidable procedure for checking validity on all possible traces.

### 3.2 Examples of Target Specifications

We now explain the examples of specifications in Table 1 and show how our grammar captures them.

The upper set of specifications in Table 1 shows examples that use only forall ( $\forall$ ) quantifiers, captured by  $\phi_{\forall}$  in our grammar. The specification for *External Consistency* of Google Spanner was described earlier. *Election Safety* of the Raft [42] protocol states that a unique leader is elected in each term. *Update Propagation* specification of Chain Replication [51] states that if a node  $n_0$  is placed before  $n_1$  in the

chain, then the commit log on  $n_1$  is a prefix ( $\preceq_u$ ) of the commit log on  $n_0$ . Predicates in these specifications are captured by Field Preds ( $P_F$ ) in our grammar. Note that the Update Propagation specification has a user-defined predicate  $\preceq_u$ .

The bottom set shows two specifications that require nested existential quantifiers, captured by  $\phi_{\forall\exists}$  in our grammar. *Whitelist Safety* of a Firewall protocol [40] states that for any received event that is allowed by the firewall, there exists a prior event that granted the sender. Note that this uses the standard happens-before [31] temporal operator ( $\prec$ ) between events, supported by Event Preds ( $P_E$ ) in our grammar. *Quorum Votes* of a Consensus protocol [39] states that any decision must have received a quorum of votes, which is supported by using an augmented existential quantifier with the SC constraint that the number of votes for a decision is greater than or equal to the number that makes a *quorum*.

### 3.3 User guidance enabled by the grammar

The grammar shown in Figure 3 enables the following user guidance that customizes the search space and identifies domain or protocol-specific relationships between events.

- (UG1) Identify interesting event combinations over which the tool should learn the specifications;
- (UG2) Provide complex predicates (e.g., checking prefix relationship between lists) that can appear in the target formula;
- (UG3) Instrument the program with *ghost code* to expose some system state as auxiliary payload fields in events.

## 4 Search Space for Target Specifications

To prepare the search space for learning target formulas, PINFER heuristically selects the quantified events  $\vec{e}_i, \vec{e}_j$ , which we call *event combinations*. Note that although it is possible, in principle, to systematically search over all possible event combinations (e.g., in increasing number of events), this would become impractical for protocols with many event types, and likely result in hard-to-understand specifications.

### 4.1 Determining event combinations

PINFER performs a lightweight static analysis of event handlers in the P program to extract causal relationships between events. Our heuristics for choosing event combinations are driven by common patterns in protocol specifications. These are summarized in the top part of Table 3, where the first column lists the causal relation pattern that results in one or more event combinations with the quantified events  $\vec{e}_i, \vec{e}_j$  listed in the last two columns, respectively. For ease of understanding, we show only the event types in the last two columns for each causal pattern. The bottom part of the same table illustrates these patterns on our running example with the listed event combinations on event types, that result

| Causal Relation Patterns                                   | Event Comb.                        |   |
|--|------------------------------------|---|
|  | $\forall \vec{e}_i : \vec{\tau}_i$ | $\exists \vec{e}_j : \vec{\tau}_j$                          |
| Same-sourced-sends<br>(sends $e_0$ , sends $e_1$ )         | $\langle \tau_0, \tau_1 \rangle$   | empty   |
| Receive-then-send<br>(receives $e_0$ , then sends $e_1$ )  | $\langle \tau_0, \tau_1 \rangle$   | empty   |
|  | $\langle \tau_1 \rangle$           | $\langle \tau_0 \rangle$                                    |
| Send-then-listen<br>(sends $e_0$ , then listens to $e_1$ ) | $\langle \tau_0, \tau_1 \rangle$   | empty   |
|  | $\langle \tau_0 \rangle$           | $\langle \tau_1 \rangle$                                    |
| <b>Running example</b>                                     | <b>Lines</b>                       |   |
|  | 21                                 | $\langle \mathcal{E}, \mathcal{E} \rangle$ empty            |
| Same-sourced-sends   | 21, 24, 26                         | $\langle \mathcal{E}, \mathcal{N} \rangle$ empty            |
|  | 17, 24, 26                         | $\langle \mathcal{N}, \mathcal{N} \rangle$ empty            |
| Receive-then-send  | 19, 21                             | $\langle \mathcal{N}, \mathcal{E} \rangle$ empty            |
|  |                                    | $\langle \mathcal{E} \rangle$ $\langle \mathcal{N} \rangle$ |
| Send-then-listen   | N/A                                | -   |

**Table 3. Causal relation patterns** that result in the listed **Event Combinations** ( $\forall e_i : \tau_i$  and  $\exists e_j : \tau_j$ ).

from causal patterns identified on the respective lines in the P program (Figure 1a).

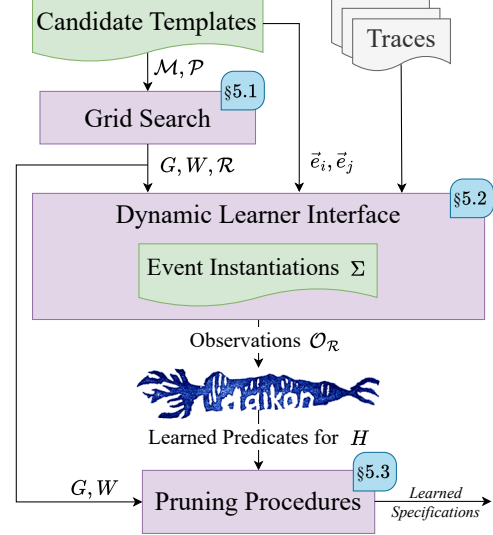
**Same-sourced-sends.** In this pattern, events with types  $\tau_0$  and  $\tau_1$  (with possibly  $\tau_0$  and  $\tau_1$  being the same types) are sent in the same entry function or event handler. This guides PINFER to learn specifications with forall quantifiers over events with types  $\tau_0$  and  $\tau_1$ .

**Receive-then-send.** A machine may send out an event of type  $\tau_0$  while handling an event of type  $\tau_1$ , where  $\tau_0 \neq \tau_1$ . This results in PINFER identifying two event combinations, as shown in the table. Notably, the second event combination, where  $\vec{e}_j = \langle \tau_0 \rangle$ , guides PINFER to learn specifications with forall quantifiers over  $e_1 : \tau_1$  and *existential* quantifiers over  $e_0 : \tau_0$ . These specifications capture triggering conditions under which events of type  $\tau_1$  can be sent, e.g., after receiving a quorum of events of type  $\tau_0$ .

**Send-then-listen.** A machine may send an event of type  $\tau_0$  and then start listening for an event of type  $\tau_1$ , where  $\tau_0 \neq \tau_1$ . Specifications learned using the listed event combinations can capture certain liveness properties, e.g., for every event of type  $\tau_0$  sent by the machine, there exists an event of type  $\tau_1$  that will be received by the machine.

## 4.2 Generating Candidate Templates

After determining the event combinations, PINFER automatically generates a candidate template  $CT(\vec{e}_i, \vec{e}_j) = (\mathcal{M}, \mathcal{P})$  for each event combination. Here,  $\mathcal{M}$  is a set of terms generated by recursively expanding the “Terms” rule in the grammar (Figure 3), but with restriction to the terms constructed from accesses to payload fields of the quantified events in  $\vec{e}_i$  and  $\vec{e}_j$ . For example, for the event combination  $\vec{e}_i = \langle e_0 : \mathcal{E} \rangle, \vec{e}_j = \langle e_1 : \mathcal{N} \rangle$  in RLE protocol,  $\mathcal{M} = \{\text{indexof}(e_0), \text{indexof}(e_1), e_0.\text{nodeId}, e_1.\text{vote}\}$ . Similarly,  $\mathcal{P}$  is a set of predicates generated by recursively expanding the “Preds” rule in the grammar (Figure 3), but with restriction



**Figure 4. PINFER search process:** Combining an enumerative Grid Search with dynamic learning by Daikon.

to the predicates over the quantified events in  $\vec{e}_i$  and  $\vec{e}_j$  and their payload fields. The generated  $\mathcal{P}$  for this example was shown earlier (Table 2, §2.5).

## 5 Learning Procedure for Specifications

The learning procedure for target specifications in PINFER combines enumerative search using the Candidate Templates, and dynamic learning from the traces using a dynamic learner such as Daikon [14]. The step-wise workflow is shown in Figure 4. In an outer loop, PINFER generates Candidate Templates  $CT(\vec{e}_i, \vec{e}_j) = (\mathcal{M}, \mathcal{P})$ , as described in the previous section.

An inner loop performs a *grid search* for each CT, to learn specifications of increasing complexity that vary in the number of predicates and terms that may appear in the target formula. In this section, we describe how the grid search works and how PINFER automatically generates inputs to Daikon in the Dynamic Learner Interface.

### 5.1 Enumerative Grid Search

To manage the search space complexity of target formulas, PINFER performs a grid search over three parameters ( $g, w, h$ ) that control the number of predicates in  $G, W$  and the number of terms in  $H$ , respectively. Specifically, for a given Candidate Template  $CT = (\mathcal{M}, \mathcal{P})$ , PINFER enumerates different choices of  $G$  and  $W$  by conjoining  $g$  and  $w$  number of predicates, respectively, from  $\mathcal{P}$ . For learning predicates in  $H$ , PINFER does not directly control the number, since  $H$  is learned by Daikon. Instead, PINFER enumerates subsets of  $\mathcal{M}$  with  $h$  terms, over which Daikon learns the predicates in  $H$ . We refer to each such subset of terms as a *Relate Set*, denoted as  $\mathcal{R}$ . PINFER depends on the capabilities of the dynamic learner to learn sufficiently expressive predicates in  $H$ . By default,

PINFER performs a search over the 3-dimensional grid space starting at  $(g = 0, w = 0, h = 1)$  and explores different grid points up to a user-settable bounds in each parameter. Note that the  $w$  parameter is skipped when learning  $\phi_V$  formulas.

## 5.2 Dynamic Learner Interface

PINFER provides a Dynamic Learner Interface to prepare inputs to the dynamic learner for learning predicates in  $H$ . First, for each event combination  $(\vec{e}_i, \vec{e}_j)$ , PINFER *filters* each given trace  $(\mathcal{T} \in \mathbb{T})$  by removing events of types that do not match  $\vec{e}_i$  or  $\vec{e}_j$ , preserving only relevant events and their temporal order.

### Quantifier Instantiations and Evaluations on Traces.

For ease of exposition, we refer to quantified events in target formulas as *symbolic* events (denoted  $\vec{e}_i, \vec{e}_j$  as before), and events that occur in a trace as *concrete* events (denoted  $\vec{z}$ ). Given the (filtered) traces and a specific choice of  $G$  and  $W$  (from the enumerative grid search), PINFER computes *instantiations*, denoted  $\sigma$ , where symbolic events  $\vec{e}_i, \vec{e}_j$  are instantiated by type-compatible concrete events  $\vec{z}_i, \vec{z}_j$ , respectively, that occur within the scope of a single trace. Given an instantiation  $\sigma$ , we can compute an *evaluation* of a term  $T$  or a predicate  $P$ , denoted  $\llbracket T \rrbracket_\sigma, \llbracket P \rrbracket_\sigma$ , respectively, by instantiating symbolic events in  $T, P$  with the corresponding concrete events in  $\sigma$ . In particular, an evaluation under  $\sigma$  results in a concrete value for a term, and a true ( $\top$ ) or false ( $\perp$ ) value for a predicate. We extend evaluations to conjunctions of predicates in  $G, W$ , denoted as  $\llbracket G \rrbracket_\sigma, \llbracket W \rrbracket_\sigma$ , respectively, which result in true/false values.

**Learning  $\phi_V$  formulas.** PINFER computes a set of instantiations, denoted  $\Sigma$ , as follows:

$$\Sigma = \bigcup_{traces} \{\sigma \mid \llbracket G \rrbracket_\sigma = \top\}$$

Intuitively,  $\Sigma$  represents all instantiations  $\sigma$  of symbolic events by concrete events in the given traces, such that  $G$  evaluates to true under  $\sigma$ . In this case, a predicate  $P_H$ , such that  $\llbracket P_H \rrbracket_\sigma$  is true under all  $\sigma \in \Sigma$ , holds under the guard  $G$  and therefore can be included in  $H$  in the target formula.

To find such predicates in  $H$ , PINFER computes a set of observations resulting from evaluations of terms in  $\mathcal{R}$  under each  $\sigma \in \Sigma$ , denoted  $\mathcal{O}_\mathcal{R}$ , as input to the dynamic learner. Recall that in the grid search, PINFER enumerates  $\mathcal{R}$  as a subset of  $h$  terms over which Daikon discovers predicates in  $H$ . As an example, suppose  $\mathcal{R} = \{e_0.a, e_1.b\}$  and PINFER computed the following observations  $\mathcal{O}_\mathcal{R}$  (where  $t \mapsto v$  indicates that term  $t$  evaluates to value  $v$  under some  $\sigma$ ):

$$\mathcal{O}_\mathcal{R} = \{\{e_0.a \mapsto 0, e_1.b \mapsto 2\}, \{e_0.a \mapsto 2, e_1.b \mapsto 3\}\}$$

Then, the dynamic learner can learn predicates  $e_0.a < e_1.b$ ,  $e_0.a \geq 0$ ,  $e_1.b > 0$ , as they hold in each observation in  $\mathcal{O}_\mathcal{R}$ .

**Learning  $\phi_{V\exists}$  formulas.** For handling the existential quantifiers, PINFER computes the set  $\Sigma$  as a set of tuples:

$$\Sigma = \bigcup_{traces} \{(\sigma_i, \omega(\sigma_i)) \mid \llbracket G \rrbracket_{\sigma_i} = \top \wedge \forall \sigma_j \in \omega(\sigma_i). \llbracket H_1 \rrbracket_{\sigma_i \cup \sigma_j} = \top\}$$

In each tuple, the first component is an instantiation  $\sigma_i$  for symbolic events  $\vec{e}_i$  such that  $G$  holds on the concrete events (similar to the  $\phi_V$  formulas above). The second component is a *set* of instantiations, denoted  $\omega(\sigma_i)$ , where each element  $\sigma_j \in \omega(\sigma_i)$  is an instantiation for symbolic events  $\vec{e}_j$ , such that  $W$  evaluates to true under the combined (union of) instantiations  $\sigma_i$  and  $\sigma_j$ . Intuitively, each  $\sigma_j \in \omega(\sigma_i)$  provides a concrete *witness* from the trace for the existentially quantified  $\vec{e}_j$ , such that predicates in  $W$  hold on the concrete events. Importantly, if there is no such witness  $\sigma_j$  for some  $\sigma_i$ , i.e.,  $\omega(\sigma_i)$  is empty, then the associated  $W$  and  $\Sigma$  are discarded, and PINFER proceeds to the next  $W$  in the grid search.

Otherwise, PINFER computes a set of observations  $\mathcal{O}_\mathcal{R}$  over which Daikon discovers predicates in  $H$ . It partitions  $\mathcal{R}$  into two disjoint subsets  $\mathcal{R}_{V\exists}$  and  $\mathcal{R}_V$ . The partition  $\mathcal{R}_{V\exists}$  contains terms  $t$  where some symbolic events from  $\vec{e}_j$  appear in  $t$ , while  $\mathcal{R}_V$  contains terms  $t$  such that no symbolic event from  $\vec{e}_j$  appears in  $t$ . Then, PINFER computes  $\mathcal{O}_\mathcal{R}$  as follows:

$$\mathcal{O}_\mathcal{R} = \{(O_i, \text{Witnesses}(O_i)) \mid (\sigma_i, \omega(\sigma_i)) \in \Sigma\}$$

Here, each observation in  $\mathcal{O}_\mathcal{R}$  is a tuple corresponding to a tuple in  $\Sigma$ . The first component  $O_i$  are evaluations of terms in  $\mathcal{R}_V$  for  $\sigma_i$ . The second component is a *non-empty* set  $\text{Witnesses}(O_i) = \{O_{j1}, O_{j2}, \dots\}$ , where each  $O_{jk}$  is an evaluation of terms in  $\mathcal{R}_{V\exists}$  under  $\sigma_i \cup \sigma_k$  where  $\sigma_k \in \omega(\sigma_i)$ .

Now, PINFER leverages a dynamic learner to find predicates  $H$  that hold over  $(O_i, O_{jk})$  for *some*  $O_{jk}$  in each tuple in  $\mathcal{O}_\mathcal{R}$  (since  $\vec{e}_j$  is existentially quantified). We demonstrate this for our running example next.

**Running example for RLE protocol.** Consider that in the grid search for the event combination  $\vec{e}_i = \langle e_0 : \mathcal{E} \rangle$  and  $\vec{e}_j = \langle e_1 : \mathcal{N} \rangle$ , PINFER has enumerated  $G = \top$ ,  $H_1 = e_1 < e_0$ , and  $\mathcal{R} = \{e_0.nodeId, e_1.vote\}$ . Since the *nodeId* of an  $\mathcal{E}$  event must have appeared earlier in  $\mathcal{N}$  events, suppose PINFER computes  $\mathcal{O}_\mathcal{R}$  for some given trace as shown below (with only one tuple shown for ease of exposition):

$$\mathcal{O}_\mathcal{R} = \left\{ \left( \{e_0.nodeId \mapsto 4\}, \left\{ \{e_1.vote \mapsto 2\}, \{e_1.vote \mapsto 4\}, \{e_1.vote \mapsto 4\} \right\} \right) \right\}$$

PINFER asks Daikon to discover predicates that relate the values from the first component and the (multi) set of values from the second component, in each tuple in  $\mathcal{O}_\mathcal{R}$ . This avoids an explicit tracking of disjunctions that would be required to find some witness in each  $\omega(\sigma_i)$ . In our example, since 4 (from the first component) is a member of the (multi) set  $\{2, 4, 4\}$  (from the second component), Daikon discovers the



*Member* relationship, i.e.,  $e_0.\text{nodeId} \in \{e_1.\text{vote}\}$ . This corresponds to learning the predicate  $H : e_0.\text{nodeId} = e_1.\text{vote}$  under an existentially quantified  $e_1$ . While *Member* limits  $H$  to an equality predicate, we found this sufficient for learning many protocol specifications with existential quantifiers.

After Daikon discovers the predicate  $e_0.\text{ballot} = e_1.\text{vote}$  in  $H$ , PINFER combines it with the enumerated  $G$  ( $\top$ ) and  $W$  ( $e_1 \prec e_0$ ) to assemble the following  $\phi_{\forall\exists}$  formula, resulting in Eqn (4):  $\forall e_0 : \mathcal{E}. \exists e_1 : \mathcal{N}. e_1 \prec e_0 \wedge e_0.\text{nodeId} = e_1.\text{vote}$ .

**Learning Set Cardinality constraints.** Recall that a Set Cardinality (SC) constraint can be used to augment existential quantification in  $\phi_{\forall\exists}$  formulas (§1.1). PINFER can also learn an SC constraint by using Daikon. We illustrate this process on an example specification for the Consensus protocol [39]. This specification states that *every decision (eDecide) receives a majority of votes (eVote)*, shown in the last row of Table 1 (§3.2). In this protocol, an *eDecide* event carries a decided *value* and a *ballot*, and an *eVote* event carries a *vote*, representing the ballot it supports. To provide the special constant in the SC constraint, we add an *eConfig* event with a field for the quorum size *quorum*, which is extracted from a system configuration. An *eConfig* event is announced (i.e., added to a trace) during the protocol startup. Note that this allows PINFER to consider traces that are generated from systems with different number of nodes, i.e., with different constants for the required quorum size, but where each trace has an event that announces the required *quorum*.

Now, when PINFER performs grid search for the event combination  $\vec{e}_i = \langle e_0 : e\text{Decide} \rangle$ ,  $\vec{e}_j = \langle e_1 : e\text{Vote} \rangle$ , it enumerates  $G = \top$  and  $W = (e_1 \prec e_0) \wedge (e_0.\text{ballot} = e_1.\text{vote})$ . According to the protocol semantics, an *eDecide* is emitted only after a majority of *eVote* events satisfying  $W$  have been sent. Therefore, for every  $\sigma_i$  that instantiates  $e_0$  with a concrete *eDecide* event, there must exist a set of witnesses  $\omega(\sigma_i)$  that instantiate  $e_1$  with concrete *eVote* events such that  $\llbracket W \rrbracket_{\sigma_i \cup \sigma_j}$  is true for all  $\sigma_j \in \omega(\sigma_i)$  and  $|\omega(\sigma_i)| \geq \text{quorum}$ . Given this, Daikon learns that the predicate  $|\omega(\sigma_i)| \geq \text{quorum}$  holds over all such witness traces. PINFER then interprets this predicate as a Set Cardinality constraint and augments the existential quantifier accordingly, as shown in Table 1. Note that the only user guidance needed is for adding the *eConfig* events with the constant *quorum* (we view this as UG3). PINFER automatically learns the shown specification, including the SC constraint for the augmented existential quantifier.

### 5.3 Formula Assembly and Pruning Procedures

After the dynamic learner has learned predicates for  $H$ , PINFER uses sanitization steps to remove those that are irrelevant or do not express relationships between quantified events. For example, comparing two HTTP status codes (integers) using  $\leq$  may get discovered as a predicate by the dynamic learner, but this is unlikely to be useful. After such predicates are removed, PINFER conjoins the remaining ones to

construct  $H$  and assembles the target formula by putting together the quantified  $\vec{e}_i, \vec{e}_j$  with the  $G, W, H$ .

**Pruning Procedures.** PINFER can often learn a large number of target formulas after dynamic learning, e.g., it assembled more than 1600 target formulas for Vertical Paxos. We implemented logic-based pruning procedures in PINFER to soundly reduce this number by eliminating tautologies, subsumed formulas, and symmetric formulas. These procedures are applied until reaching a fixpoint, resulting in a set of subsumption-free formulas that are reported to the user. We describe them briefly below (details in the Appendix, §C).

**Pruning by syntactic checking.** PINFER abstracts the predicates in  $G, H$  to propositional variables  $\mathcal{G}, \mathcal{H}$ , respectively. Then, if  $\mathcal{H} \subseteq \mathcal{G}$ , then  $G \rightarrow H$ . PINFER uses this check to prune tautologies and to detect subsumptions between two  $\phi_{\forall}$  formulas with the same quantified events.

**Pruning by semantic checking.** To detect implications  $G \rightarrow H$  missed by syntactic checking, PINFER leverages the Z3 [12] SMT solver to eliminate tautologies and subsumptions (with uninterpreted user-defined predicates/functions.)

**Pruning symmetric formulas.** Two  $\phi_{\forall}$  formulas are equivalent by symmetry if they have the same quantified variables, and one can be obtained from the other via rewriting symmetric predicates (e.g.,  $e_0.a < e_1.b$  to  $e_0.b > e_1.a$ ).

**Pruning by PChecker.** Although the learned formulas are consistent with the given traces, they may contain falsifiable formulas when considering all behaviors of the protocol. We can (optionally) generate P specifications from the learned formulas and check them on the P program by utilizing PChecker [13] to eliminate formulas that are reported false.

## 6 Evaluations

Our evaluation addresses the following research questions:

**RQ1:** Can Pinfer effectively learn protocol specifications that developers consider essential correctness properties?

**RQ2:** How effectively do pruning procedures reduce the number of learned specifications?

**RQ3:** Do the specifications learned by PINFER help deductive verification, particularly for learning inductive invariants?

**RQ4:** How does PINFER compare to existing approaches? We evaluated PINFER against state-of-the-art tools SWISS [20] and DuoAI [57] on a complex benchmark for comparison.

### 6.1 Implementation

We implemented PINFER as an extension of the P compiler [3, 13], with approximately 7,000 lines of code in C#. We implemented a Dynamic Learner Interface (§5.2) with about 1,400 lines of code in Java to inter-operate with Daikon [14]. We use the Z3 SMT solver [12] for semantic checking in pruning. As a specialized mode within P, PINFER accepts formal models from P users, autonomously executes the model checker to generate event traces, and subsequently presents the learned specifications to the user.

**Table 4. Results for learning protocol specifications (RQ1).** The LoC column reports the lines of code in the P model.  $S_{goals}$  shows the number of protocol specifications in the challenge set, and a ✓ indicates that all are learned by PINFER. The next column provides a description (and if  $\forall\exists$  is needed). The last column shows the end-to-end run time of PINFER (in seconds).

| Benchmark                          | LoC  | $S_{goals}$ | Description of Specifications  | Time (s) |
|------------------------------------|------|-------------|--|----------|
| Ring Leader Election [10]          | 47   | 3 ✓         | Unique leader, Leader Highest ID, Leader Voted   | 352      |
| Consensus <sup>†</sup> [28]        | 61   | 1 ✓         | Safety (Unique Decision)   | 925      |
| 2PC [3]                            | 238  | 2 ✓         | Atomicity ( $\forall\exists_n$ ), Safety (Commit or Abort)   | 3935     |
| Sharded KV <sup>†*</sup> [28]      | 48   | 1 ✓         | Safety (Unique Shard Owner)  | 481      |
| Paxos [33]                         | 164  | 2 ✓         | Safety (Unique Decision), B3( $\mathcal{B}$ ) <sup>†</sup>   | 1155     |
| Distributed Lock <sup>†</sup> [39] | 73   | 1 ✓         | Safety (Mutual Exclusion)  | 902      |
| Vertical Paxos [35]                | 241  | 2 ✓         | Safety (Unique Decision), B3( $\mathcal{B}$ ) <sup>†</sup>   | 2589     |
| Firewall [40]                      | 80   | 1 ✓         | Whitelisted Safety ( $\forall\exists$ )  | 753      |
| Lock Server <sup>†</sup> [39]      | 102  | 1 ✓         | Safety (Mutual Exclusion)  | 620      |
| Chain Replication [51]             | 452  | 6 ✓         | Update Propagation, <sup>‡</sup> Inprocess Requests, <sup>‡</sup> Linearizability ( $\forall\exists$ ) <sup>†#</sup>     | 984      |
| Raft <sup>*†</sup> [42]            | 1191 | 5 ✓         | Election Safety, State Machine Safety, Log Matching <sup>+</sup><br>Leader Append-Only, Leader Completeness <sup>+</sup> | 4223     |
| GlobalClock                        | 177  | 3 ✓         | Clock Monotonicity, Real-time ordering <sup>#</sup>  | 805      |
| DBLeaderElection                   | 1178 | 5 ✓         | Monotonic Commits, Unique Leader, Read Consistency <sup>†#</sup>   | 697      |
| MVCC-2PC                           | 1135 | 10 ✓        | Atomicity ( $\forall\exists_n$ ), <sup>#</sup> Snapshot Isolation ( $\forall\exists$ ) <sup>‡#</sup>                     | 6642     |

<sup>†</sup> P model translated from an IVy model

\* Sharded KV models key transfers but not modifications, Raft omits cluster reconfiguration

<sup>†</sup> Requires user-specified event combinations (UG1)

<sup>+</sup> Requires user-defined predicate over log entries exposed to traces via events (UG2 & UG3)

<sup>‡</sup> Requires events that expose commit logs to traces <sup>#</sup> Implied by a subset of the learned specifications

## 6.2 Benchmarks

We evaluated PINFER on a diverse set of benchmarks, including 11 well-known distributed protocols from past literature and 3 real-world industrial case-studies.

**Well-known protocols.** We hand-translated 4 protocol models (marked with <sup>†</sup> in the “Benchmark” column of Table 4) from Ivy to P that were considered as benchmarks in previous papers, namely, IC3FOL [2, 28], SWISS [20], and I4 [1, 39]. In addition, we developed P models for 5 foundational protocols: Two-Phase Commit (2PC) [18], Ring Leader Election protocol [10], Paxos [33], Vertical Paxos [35] and Raft [42]. Notably, our Raft model is fully functional (excluding cluster reconfiguration) and includes two failure models: unreliable network communications and node crashes.

**Real-world case studies.** To demonstrate the applicability of PINFER on real-world case studies, we considered three protocol models that were developed by service teams at AnonCompany. GlobalClock models a distributed clock synchronization service, DBLeaderElection models a distributed leader election protocol that uses an underlying append-only log to build consensus, and finally, MVCC-2PC models a distributed transactional service that combines multi-version

concurrency control [7] with two-phase commit [18] to guarantee atomicity and snapshot isolation. The formal models and specifications were written by the developers, we demonstrate that PINFER can automatically learn these specification that the teams had written when building these services.

**Setup.** The experiments were conducted on an AMD EPYC 7532 32-core CPU with 1TB of memory. For each benchmark, input traces are generated using P model checker [13], which systematically explores nondeterminism in system configurations and event interleavings. For each benchmark, we used 3–5 different system configurations, i.e., number of processes/inputs or failures. For example, for Paxos, we varied the numbers of proposers, acceptors, and learners. We generated a total of 10,000 traces to capture diverse protocol behaviors. In the grid search, we set the parameter upper bounds as  $g = 2$ ,  $w = 2$ , and  $h = 2$ , i.e., allowing at most two atomic predicates in  $G$ ,  $W$  and up to two terms in the Relate Set  $\mathcal{R}$  for  $H$ . These parameter settings are based on our observation of common patterns in protocol specifications, but remain user-adjustable.

### 6.3 RQ1: Learning protocol specifications

To address RQ1, we establish a challenge set comprising 43 protocol specifications (total of  $S_{goals}$  column in Table 4): those derived from seminal papers that introduced the protocols in our benchmarks [10, 28, 33, 35, 39, 42, 44, 51], and for industrial case studies, specifications authored by the developers themselves. We designate these specifications as *goal specifications* for our evaluation. The complete set of first-order logic (FOL) formulas constituting these specifications is presented in Appendix A. Notably, 5 of these protocol specifications require  $\forall\exists$ -quantifiers, including 2 with augmented existential quantifiers. To determine whether a protocol specification has been successfully identified, we employ a light-weight semantic entailment checking (part of our pruning procedures), since the learned specifications may be syntactically different from the original specifications while being semantically equivalent.

Table 4 presents an overview of our benchmarks and summarizes the results. The LoC column indicates the protocol complexity by reporting the lines of code in each P model. Subsequent columns detail the goal specifications ( $S_{goals}$ ) for each benchmark, provide descriptions of these protocol specifications, and document the end-to-end execution time of PINFER. Footnotes in the table denote instances where user guidance (UG) was necessary to successfully learn the goal specifications. *Notably, PINFER successfully learned all 43 protocol specifications that we identified in the challenge set as goal specifications ( $S_{goals}$ ), with 28 of these learned automatically without any user intervention.* The remaining 15 required some degree of user guidance. We elucidate the key features of PINFER and the user inputs that proved essential for learning the remaining protocol specifications.

**When is user guidance needed?** PINFER allows three kinds of user guidance (§3.3). In our experiments, learning protocol specifications that require UG1 are marked with † in Table 4. For UG1, users leverage their understanding of the protocol to define crucial event combinations, especially, those overlooked by PINFER’s heuristics based on causally-related events. For instance, in the Chain Replication benchmark, we specified an event combination of a read response and a write response to learn 2 of the *Linearizability* specifications. UG2 is needed when predicates on custom data types are important, since generic dynamic learners do not usually support such predicates. UG3 is more nuanced, as nodes maintain various local states, and users may have to *selectively* expose states to yield meaningful specifications. We found that exposing local states reflecting *shared data*, e.g., commit logs to a shared store, is especially useful. Since protocols aim to satisfy certain consistency models over shared data across nodes, relationships over these states are often reflected in protocol specifications. In our evaluations, learning protocol specifications marked with + in Table 4 require UG2 and UG3. For instance, we instrumented the P model of Chain

**Table 5. Results showing the effect of pruning (RQ2).**

$S_{raw}$ ,  $S_{asm}$ ,  $S_{syn}$  and  $S_{sem}$  columns show the number of specifications after dynamic learning with Daikon, after formula assembly, after pruning with syntactic checking and with semantic checking, respectively. The last number ( $S_{sem}$ ) is the same as the number of learned specifications in PINFER’s outputs. *RR* column shows the reduction ratio ( $S_{raw}/S_{sem}$ ), and  $S_{\perp}$  shows the number of learned specifications falsified by PChecker.

| Benchmark         | $S_{raw}$ | $S_{asm}$ | $S_{syn}$ | $S_{sem}$ | RR  | $S_{\perp}$ |
|-------------------|-----------|-----------|-----------|-----------|-----|-------------|
| Ring Leader Elec. | 479       | 183       | 52        | 30        | 15× | 0           |
| Consensus         | 361       | 92        | 28        | 28        | 13× | 1           |
| 2PC               | 887       | 125       | 60        | 46        | 19× | 9           |
| Sharded KV        | 289       | 93        | 23        | 19        | 15× | 0           |
| Paxos             | 850       | 229       | 52        | 49        | 17× | 5           |
| Distributed Lock  | 740       | 272       | 91        | 77        | 9×  | 0           |
| Vertical Paxos    | 1646      | 514       | 104       | 94        | 17× | 5           |
| Firewall          | 606       | 160       | 46        | 40        | 19× | 4           |
| Lock Server       | 399       | 126       | 44        | 35        | 11× | 0           |
| Chain Replication | 533       | 116       | 40        | 37        | 14× | 2           |
| Raft              | 1080      | 415       | 73        | 58        | 18× | 10          |
| GlobalClock       | 368       | 136       | 37        | 37        | 9×  | 0           |
| DBLeaderElect.    | 577       | 192       | 50        | 47        | 12× | 9           |
| MVCC-2PC          | 5789      | 1491      | 334       | 236       | 24× | 0           |

Replication by adding an event that exposes commit logs, and provided a user-defined predicate  $\preceq_u$  over log sequences, where  $l_i \preceq_u l_j$  means that  $l_i$  is a prefix of  $l_j$ . In future work, we would explore how these predicates can be generated automatically using program synthesis techniques [4, 19].

### 6.4 RQ2: Effect of pruning strategies

We evaluate the effectiveness of the pruning procedures using the same setup as RQ1. Table 5 shows the number of specifications remaining after each pruning step. The last two columns report the Reduction Ratio (RR) and the numbers of specifications falsified with PChecker ( $S_{\perp}$ ).

The reported number of learned specifications ( $S_{sem}$ ) is under a hundred for all benchmarks except MVCC-2PC. The higher count in the MVCC-2PC benchmark is due to an extensive search space (with 62k invocations of Daikon during grid search). A substantial reduction from hundreds of specifications in  $S_{raw}$  is achieved by the pruning steps, with a geometric average reduction ratio of 14×. Importantly, the pruning steps did not eliminate any challenge protocol specifications. We note that pruning by efficient syntactic checking in PINFER eliminates most redundancies, drastically reducing the number of SMT queries used for semantic checking. On average, the pruning steps up to  $S_{sem}$  finished within 7 seconds, and the PChecker for optional falsification ( $S_{\perp}$ ) was run with a 10-hour timeout.

**Table 6. Results for learning inductive invariants using PINFER (RQ3).** We use a subset of benchmarks with proofs in PVerifier [40].  $I_e$  and  $I_s$  columns show the numbers of event-based and state-based inductive invariants, respectively.

| Benchmark            | $I_e$ | $I_s$ | All learned |
|----------------------|-------|-------|-------------|
| Ring Leader Election | 3     | 0     | ✓           |
| Consensus*           | 1     | 5     | ✓           |
| Distributed Lock*    | 4     | 4     | ✓           |
| Lock Server*         | 4     | 4     | ✓           |
| Firewall             | 1     | 0     | ✓           |
| Sharded KV*          | 1     | 1     | ✓           |

### 6.5 RQ3: Learning inductive invariants

To evaluate PINFER’s utility for downstream verification tasks, we conducted experiments with benchmarks previously verified in PVerifier [40]—a framework for deductive verification of P programs. Our evaluation focused specifically on the *inductive invariants* required by PVerifier proofs, many of which match the grammar of the target specifications learned by PINFER.<sup>2</sup> These inductive invariants fall into two categories: event-based and state-based, with the latter requiring additional user-defined instrumentation (mainly for adding events that expose local machine states).

Table 6 presents our results for learning inductive invariants, with  $I_e$  and  $I_s$  representing the number of event-based and state-based invariants required for correctness proofs, respectively. With user-defined instrumentation (for benchmarks marked with asterisks \*), PINFER successfully learns all inductive invariants. Without such instrumentation, PINFER still learns all event-based invariants ( $I_e$  only) but fails to learn the state-based ones (not  $I_s$ ).

### 6.6 RQ4: Comparison with other tools

A key challenge for invariant learning tools is their ability to handle large, complex distributed protocols. To evaluate this capability, we compared PINFER with SWISS [20] and DuoAI [57] on Vertical Paxos, the most computationally demanding open-source benchmark for PINFER. Using a 16-core machine with 32GB of memory, we tested all three approaches on the IVy model of Vertical Paxos. DuoAI failed to complete due to an implementation bug (which we have reported to the developers). SWISS did not produce a solution in 24 hours, probably because it searches for complete proofs in what becomes a prohibitively large search space – Vertical Paxos invariants include up to eight literals – while its automated verifier struggles with complex queries. In contrast, PINFER successfully learned both safety correctness specifications and additional (likely) invariants for Vertical Paxos in about 40 minutes.

<sup>2</sup>Exact formulas are provided in Appendix A.

## 7 Related Work

PINFER is most closely related to invariant learning approaches, based on use of verifiers or dynamic learning.

**Verifier-aided invariant synthesis.** SWISS [20], DuoAI [57] are state-of-the-art tools for learning inductive invariants (with  $\forall\exists$ -quantifiers) to formally prove safety properties using verifiers [44, 52]. Although powerful, these approaches face several limitations: they require user-provided safety properties and source models expressed in decidable fragments of FOL [43]. Their applicability in practice is limited by both verifier performance and scalability. This became evident in our comparative evaluation (§6.6), where SWISS failed to terminate (i.e., failed to learn invariants) for the Vertical Paxos protocol in 24 hours. Various earlier efforts [16, 27, 28, 39, 45, 58] have similar limitations, and some do not support nested existential quantifiers.

**Data-driven invariant learning** Dinv [17], LIDO [54] are leading efforts that learn safety properties from state traces, i.e., traces of global system states. Though these approaches scale better than verifier-based approaches, they have other significant limitations: neither supports  $\forall\exists$ -quantifiers. Furthermore, they require user intervention – Dinv needs user annotations of system states for invariant discovery, and without careful user guidance, it can generate millions of candidate invariants on complex protocols like Raft [42], necessitating manual refinement of annotations [17].

Other earlier efforts [8, 29, 55, 56] focused on learning *property patterns* over execution traces or message sequence charts. Specifically, Perracotta [56] uses pre-defined property templates to recognize patterns, Beschastnikh et al. [8] is tailored for finding temporal relationships between events, and Kumar et al. [29] uses a set of regular expression templates to find message sequence patterns. Although their target temporal patterns can be viewed in terms of quantifiers and precedence over the *occurrence* of events, they do not support expressing any relationships between *payloads* of the related events. On the other hand, temporal relations targeted by these techniques can be learned by PINFER.

**Verification of distributed systems** Our work is more broadly related to extensive work on formal verification of distributed systems. Efforts based on program logics and theorem-proving [38, 47, 48, 53], using interactive theorem-provers such as Rocq and Iris [26, 50], often require high manual effort to formulate and find correctness proofs. Semi-automated efforts [21, 22, 36, 40, 43, 49, 59] leverage user-provided annotations to formulate verification tasks handled by automated verifiers such as Ivy [44, 52], Dafny [37], or Uclid [30]. Among these, a recent work Kondo [59] proposes an invariant taxonomy that classifies invariants as Regular Invariants (facts about low-level details, e.g., the network) and Protocol Invariants (behaviors of the protocol). This serves as guidance for users to specify (inductive) invariants that draw connections between the network and host states.

In PINFER, these invariants can be expressed as event-based invariants when host states are made visible in event traces.

Model checking and automatic state exploration have also been used to find bugs in distributed systems [13, 24, 32], although their scalability is limited to small systems. PINFER can utilize traces generated by these techniques, and also use them to potentially falsify the learned specifications.

## 8 Conclusion

We present PINFER, an automated learning framework that learns specifications from event traces of distributed systems. It uses a specialized formula template to express target specifications with quantified events and relationships between their payloads, including support for  $\forall\exists$ -quantifiers. PINFER uses a novel learning procedure in which a static enumerative search on target formulas is combined effectively with dynamic learning on event traces. Our evaluations demonstrate that PINFER can learn all the protocol specifications from our benchmark suite of 11 well-known distributed protocols and 3 proprietary industrial-scale protocols. We additionally show that PINFER can also learn inductive invariants that benefit downstream verification tasks.

## References

- [1] GitHub - GLaDOS-Michigan/I4: The code base for the I4 prototype, as described in the SOSP '19 paper "I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols" — github.com. <https://github.com/GLaDOS-Michigan/I4>. [Accessed 11-04-2025].
- [2] GitHub - jrkoenig/folseparators: First Order Logic Separators — github.com. <https://github.com/jrkoenig/folseparators>. [Accessed 11-04-2025].
- [3] GitHub - p-org/P: The P programming language. — github.com. <https://github.com/p-org/P>. [Accessed 18-03-2025].
- [4] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.
- [5] Wei An, Xiao Bi, Guanting Chen, Shanhuang Chen, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Wenjun Gao, Kang Guan, Jianzhong Guo, Yongqiang Guo, Zhe Fu, Ying He, Panpan Huang, Jiashi Li, Wenfeng Liang, Xiaodong Liu, Xin Liu, Yiyuan Liu, Yuxuan Liu, Shanghao Lu, Xuan Lu, Xiaotao Nie, Tian Pei, Junjie Qiu, Hui Qu, Zehui Ren, Zhangli Sha, Xuecheng Su, Xiaowen Sun, Yixuan Tan, Minghui Tang, Shiyu Wang, Yaohui Wang, Yongji Wang, Ziwei Xie, Yiliang Xiong, Yanhong Xu, Shengfeng Ye, Shuiping Yu, Yukun Zha, Liyue Zhang, Haowei Zhang, Mingchuan Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, and Yuheng Zou. Fire-flyer ai-hpc: A cost-effective software-hardware co-design for deep learning, 2024.
- [6] Robert Beers. Pre-rtl formal verification: An intel experience. In *2008 45th ACM/IEEE Design Automation Conference*, pages 806–811, 2008.
- [7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.
- [8] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. Mining temporal invariants from partially ordered logs. In *Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML '11, New York, NY, USA, 2011. Association for Computing Machinery.
- [9] Marc Brooker and Ankush Desai. Systems correctness practices at aws: Leveraging formal and semi-formal methods. *Queue*, 22(6):79–96, February 2025.
- [10] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.
- [11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google’s globally-distributed database. In *OSDI*, 2012.
- [12] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’08/ETAPS’08*, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- [13] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13*, page 321–332, New York, NY, USA, 2013. Association for Computing Machinery.
- [14] 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. *Sci. Comput. Program.*, 69(1–3):35–45, December 2007.
- [15] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, July 1997.
- [16] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*, page 131–150, Berlin, Heidelberg, 2021. Springer-Verlag.
- [17] 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*, page 1149–1159, New York, NY, USA, 2018. Association for Computing Machinery.
- [18] Jim Gray. *The transaction concept: virtues and limitations*, page 140–150. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
- [19] Sumit Gulwani. Programming by examples: applications, algorithms, and ambiguity resolution. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, PPDP ’17*, page 2, New York, NY, USA, 2017. Association for Computing Machinery.
- [20] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. Finding invariants of distributed systems: It’s a small (enough) world after all. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 115–131. USENIX Association, April 2021.
- [21] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 911–929, Boston, MA, July 2023. USENIX Association.
- [22] 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*, page 1–17,



- New York, NY, USA, 2015. Association for Computing Machinery.
- [23] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
  - [24] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.
  - [25] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, page 637–650, New York, NY, USA, 2015. Association for Computing Machinery.
  - [26] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, page 637–650, New York, NY, USA, 2015. Association for Computing Machinery.
  - [27] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzkzy, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *J. ACM*, 64(1), March 2017.
  - [28] 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 2020, page 703–717, New York, NY, USA, 2020. Association for Computing Machinery.
  - [29] Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. Inferring class level specifications for distributed systems. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE ’12, page 914–924. IEEE Press, 2012.
  - [30] Shuvendu K. Lahiri and Sanjit A. Seshia. The uclid decision procedure. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 475–478, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
  - [31] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
  - [32] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.
  - [33] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
  - [34] Leslie Lamport. Industrial Use of TLA+ — lamport.azurewebsites.net. <https://lamport.azurewebsites.net/tla/industrial-use.html>, 2019. [Accessed 10-03-2025].
  - [35] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC ’09, page 312–313, New York, NY, USA, 2009. Association for Computing Machinery.
  - [36] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP ’24, page 438–454, New York, NY, USA, 2024. Association for Computing Machinery.
  - [37] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR’10, page 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.
  - [38] 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, New York, NY, USA, 2016. Association for Computing Machinery.
  - [39] 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, page 370–384, New York, NY, USA, 2019. Association for Computing Machinery.
  - [40] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. Message chains for distributed system verification. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.
  - [41] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.
  - [42] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.
  - [43] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.
  - [44] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. *SIGPLAN Not.*, 51(6):614–630, June 2016.
  - [45] Oded Padon, James R. Wilcox, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken. Induction duality: primal-dual search for invariants. *Proc. ACM Program. Lang.*, 6(POPL), January 2022.
  - [46] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE ’07, page 571–572, New York, NY, USA, 2007. Association for Computing Machinery.
  - [47] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.
  - [48] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: a separation-logic library for verifying distributed systems. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP ’23, page 113–129, New York, NY, USA, 2023. Association for Computing Machinery.
  - [49] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f\*. *SIGPLAN Not.*, 51(1):256–270, January 2016.
  - [50] The Coq Development Team. The coq proof assistant – <https://doi.org/10.5281/zenodo.14542673>, September 2024.
  - [51] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI’04, page 7, USA, 2004. USENIX Association.
  - [52] James R. Wilcox, Yotam M. Y. Feldman, Oded Padon, and Sharon Shoham. mypyvy: A research platform for verification of transition systems in first-order logic. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II*, page 71–85, Berlin, Heidelberg, 2024. Springer-Verlag.
  - [53] 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, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.
  - [54] Yuan Xia, Deepayan Sur, Aabha Shailesh Pingle, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Srivatsan Ravi. Discovering likely invariants for distributed systems through runtime monitoring and learning. In Krishna Shankaranarayanan, Sriram Sankaranarayanan, and Ashutosh Trivedi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 3–25, Cham, 2025. Springer Nature

Switzerland.

- [55] Maysam Yabandeh, Marco Canini, Dejan Kostic, and Abhishek Anand. Finding Almost-Invariants in Distributed Systems . In *Reliable Distributed Systems, IEEE Symposium on*, pages 177–182, Los Alamitos, CA, USA, October 2011. IEEE Computer Society.
- [56] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, page 282–291, New York, NY, USA, 2006. Association for Computing Machinery.
- [57] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 485–501, Carlsbad, CA, July 2022. USENIX Association.
- [58] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-Driven automated invariant learning for distributed protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 405–421. USENIX Association, July 2021.
- [59] Tony Nuda Zhang, Travis Hance, Manos Kapritsos, Tej Chajed, and Bryan Parno. Inductive invariants that spark joy: Using invariant taxonomies to streamline distributed protocol proofs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 837–853, Santa Clara, CA, July 2024. USENIX Association.
- [60] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s), September 2022.

## Appendix

### A Specifications in the challenge set

Here, we list the protocol specifications in the challenge set for each benchmark. Event names are abbreviated as letters.

#### A.1 Ring Leader Election protocol

##### Event types

- $eElectedAsLeader: \mathcal{B}$
- $eNominate: \mathcal{N}$

##### Specification: Unique Leader

$$\forall e_0, e_1 : \mathcal{B}. e_0.nodeId = e_1.nodeId$$

##### Inductive Invariants

$$\forall e_0, e_1 : \mathcal{N}. e_0.nxt = e_1.nxt \wedge e_0 \prec e_1 \rightarrow e_0.vote \leq e_1.vote$$

$$\forall e_0 : \mathcal{B}, e_1 : \mathcal{N}. e_0.nodeId \geq e_1.vote$$

$$\forall e_0, e_1 : \mathcal{N}. e_0.vote = e_0.nxt \rightarrow e_0.vote \geq e_1.vote$$

#### A.2 2PC

##### Event types

- $eCommitTxn: \mathcal{C}$
- $eAbortTxn: \mathcal{A}$
- $ePrepareSuccess: \mathcal{P}$

##### Specification: Atomicity

$$\forall e_0 : \mathcal{C} \exists e_1 : \mathcal{P}. e_0.txnId = e_1.txnId$$

##### Specification: Commit or Abort

$$\forall e_0 : \mathcal{A}, e_1 : \mathcal{C}. e_0.txnId \neq e_1.txnId$$

#### A.3 Sharded KV

##### Event types

- $eOwns^3: \mathcal{O}$
- $eTransfer: \mathcal{T}$

##### Specification: Unique Shard Owner

$$\forall e_0, e_1 : \mathcal{O}. e_0.key = e_1.key \rightarrow e_0.value = e_1.value \wedge e_0.node = e_1.node$$

##### Inductive Invariants

$$\forall e_0 : \mathcal{O}, e_1 : \mathcal{T}. e_0.key = e_1.key \rightarrow$$

$$e_0.node \neq e_1.src$$

$$\forall e_0, e_1 : \mathcal{T}. e_0.key = e_1.key \rightarrow$$

$$e_0.dst = e_1.dst \wedge e_0.value = e_1.value$$

#### A.4 Paxos

##### Event types

- $eLearn: \mathcal{D}$
- $eAcceptReq: \mathcal{A}$

##### Specification: Unique Decision

$$\forall e_0, e_1 : \mathcal{D}. e_0.value = e_1.value$$

##### Specification: B2( $\mathcal{B}$ )

$$\forall e_0 : \mathcal{D}, e_1 : \mathcal{A}. e_0.ballot < e_1.ballot \rightarrow e_0.value = e_1.value$$

#### A.5 Distributed Lock

##### Event types

- $eHasLock: \mathcal{L}$
- $eNodeState^3: \mathcal{N}$
- $eTransfer: \mathcal{T}$

##### Specification: Mutual Exclusion

$$\forall e_0, e_1 : \mathcal{L}. e_0.epoch = e_1.epoch \rightarrow e_0.node = e_1.node$$

##### Inductive Invariants

$$\forall e_0, e_1 : \mathcal{N}. e_0.hasLock \wedge e_0.epoch = e_1.epoch \rightarrow e_0.node = e_1.node$$

$$\forall e_0, e_1 : \mathcal{T}. e_0.round = e_1.round \rightarrow$$

$$e_0.node = e_1.node \wedge e_0.epoch = e_1.epoch$$

$$\forall e_0 : \mathcal{N}. e_0.hasLock \rightarrow \exists e_1 : \mathcal{T}. e_1 \prec e_0 \wedge e_0.round = e_1.round \wedge e_0.node = e_1.node \wedge e_0.epoch = e_1.epoch$$

$$\forall e_0 : \mathcal{L}, e_1 : \mathcal{N}. e_0.round = e_1.round \wedge \neg e_1.hasLock \rightarrow e_0.epoch > e_1.epoch$$

$$\forall e_0 : \mathcal{L}, e_1 : \mathcal{N}. e_0.round = e_1.round \rightarrow e_0.epoch \geq e_1.epoch$$

$$\forall e_0 : \mathcal{N}, e_1 : \mathcal{T}. e_1 \prec e_0 \wedge e_0.hasLock \rightarrow e_0.epoch \geq e_1.epoch$$

$$\forall e_0 : \mathcal{N}, e_1 : \mathcal{T}. e_0 \prec e_1 \rightarrow e_0.epoch < e_1.epoch$$

$$\forall e_0, e_1 : \mathcal{T}. e_0 \prec e_1 \rightarrow e_1.epoch < e_1.epoch$$

#### A.6 Vertical Paxos

##### Event types

- $eDecided: \mathcal{D}$
- $ePropose: \mathcal{P}$

##### Specification: Unique Decision

$$\forall e_0, e_1 : \mathcal{D}. e_0.value = e_1.value$$

##### Specification: B3( $\mathcal{B}$ )

$$\forall e_0 : \mathcal{D}, e_1 : \mathcal{P}. e_0.round < e_1.round \rightarrow e_0.value = e_1.value$$

<sup>3</sup> Added via UG3

## A.7 Firewall

### Event types

- *Sent*:  $\mathcal{S}$
- *Allowed*:  $\mathcal{A}$
- *SentFromInternal*:  $\mathcal{I}$

#### Specification: Whitelist Safety

$$\forall e_0 : \mathcal{S}. e_0.toInternal \rightarrow \exists e_1 : \mathcal{I}. e_1 \prec e_0 \wedge e_0.src = e_1.dst$$

#### Inductive Invariant

$$\forall e_0 : \mathcal{A}. \exists e_1 : \mathcal{I}. e_1 \prec e_0 \wedge e_0.node = e_1.dst$$

## A.8 Lock Server

### Event types

- *eHoldsLock*:  $\mathcal{L}$
- *eServerState*<sup>3</sup>:  $\mathcal{S}$
- *eGrant*:  $\mathcal{G}$
- *eUnlock*:  $\mathcal{U}$

#### Specification: Mutual Exclusion

$$\forall e_0, e_1 : \mathcal{L}. e_0.epoch = e_1.epoch \rightarrow e_0.node = e_1.node$$

#### Inductive Invariants

$$\forall e_0, e_1 : \mathcal{G}. e_0.epoch = e_1.epoch \rightarrow e_0.node = e_1.node$$

$$\forall e_0, e_1 : \mathcal{U}. e_0.epoch = e_1.epoch \rightarrow e_0.node = e_1.node$$

$$\forall e_0 : \mathcal{L}, e_1 : \mathcal{G}. e_0.epoch = e_1.epoch \rightarrow e_1 \prec e_0$$

$$\forall e_0 : \mathcal{S}, e_1 : \mathcal{G}. e_0.epoch = e_1.epoch \wedge e_1 \prec e_0 \rightarrow \neg e_0.holdsLock$$

$$\forall e_0 : \mathcal{S}, e_1 : \mathcal{U}. e_0.epoch = e_1.epoch \wedge e_1 \prec e_0 \rightarrow \neg e_0.holdsLock$$

$$\forall e_0 : \mathcal{S}, e_1 : \mathcal{L}. e_0.epoch = e_1.epoch \wedge e_1 \prec e_0 \rightarrow \neg e_0.holdsLock$$

$$\forall e_0 : \mathcal{L}, e_1 : \mathcal{U}. e_0.epoch = e_1.epoch \rightarrow e_0 \prec e_1$$

$$\forall e_0 : \mathcal{G}, e_1 : \mathcal{U}. e_0.epoch = e_1.epoch \rightarrow e_0 \prec e_1$$

## A.9 ChainReplication

### Event types

- *eReadSuccess*:  $\mathcal{R}$
- *eWriteResponse*:  $\mathcal{W}$
- *eNotifyLog*<sup>3</sup>:  $\mathcal{N}$

#### Specification: Update Propagation

$$\forall e_0, e_1 : \mathcal{N}. e_0.pos \geq e_1.pos \wedge e_0.epoch = e_1.epoch \rightarrow e_0.log \preceq e_1.log$$

Here,  $e_0.log \preceq e_1.log$  denotes the prefix relation:  $e_0.log$  is a prefix of  $e_1.log$ .

#### Specification: Inprocess Requests

$$\forall e_0, e_1 : \mathcal{N}. e_0.pos \leq e_1.pos \wedge e_0.epoch = e_1.epoch \rightarrow e_0.log = e_0.sent \oplus e_1.log$$

Here,  $e_0.sent \oplus e_1.log$  denotes the ordered union of requests in  $e_0.sent$  and  $e_1.log$  (by time of occurrence).

#### Specification: Linearizability

$$\forall e_0, e_1 : \mathcal{R}. e_0 \prec e_1 \wedge e_0.key = e_1.key \rightarrow e_0.version \leq e_1.version$$

$$\forall e_0 : \mathcal{R}, e_1 : \mathcal{W}. e_1 \prec e_0 \wedge e_0.key = e_1.key \rightarrow e_0.version \geq e_1.version$$

$$\forall e_0 : \mathcal{R}. \exists e_1 : \mathcal{W}. e_1 \prec e_0 \wedge e_0.version = e_1.version \wedge e_0.key = e_1.key \wedge e_0.value = e_1.value$$

$$\forall e_0, e_1 : \mathcal{W}. e_0 \prec e_1 \wedge e_0.key = e_1.key \rightarrow e_0.version \leq e_1.version$$

## A.10 Raft

### Event types

- *eBecomeLeader*:  $\mathcal{B}$
- *eEntryApplied*:  $\mathcal{A}$
- *eNotifyLog*<sup>3</sup>:  $\mathcal{N}$

#### Specification: Election Safety

$$\forall e_0, e_1 : \mathcal{B}. e_0.term = e_1.term \rightarrow e_0.leader = e_1.leader$$

#### Specification: State Machine Safety

$$\forall e_0, e_1 : \mathcal{A}. e_0.index = e_1.index \rightarrow e_0.payload = e_1.payload$$

#### Specification: Leader Append-Only

$$\forall e_0, e_1 : \mathcal{A}. e_0.term < e_1.term \rightarrow e_0.index < e_1.index$$

#### Specification: Leader Completeness

$$\forall e_0, e_1 : \mathcal{B}. e_0.term < e_1.term \rightarrow e_0.log \preceq e_1.log$$

#### Specification: Log Matching

$$\forall e_0, e_1 : \mathcal{N}. \forall i, j. e_0.log[i] = e_1.log[i] \wedge j \in [0, i) \rightarrow e_0.log[j] = e_1.log[j]$$

## A.11 GlobalClock

### Event types

- *eLocalClock*:  $\mathcal{L}$

**Specification: Clock Monotonicity**

$$\forall e_0, e_1 : \mathcal{L}. e_0.\text{latest} < e_1.\text{latest} \rightarrow e_0.\text{trueTime} < e_1.\text{trueTime}$$
**Specification: Real-time ordering**

$$\begin{aligned} \forall e_0, e_1 : \mathcal{L}. e_0.\text{trueTime} < e_1.\text{trueTime} \rightarrow e_0.\text{earliest} < e_1.\text{latest} \\ \forall e_0, e_1 : \mathcal{L}. e_0.\text{trueTime} < e_1.\text{trueTime} \\ \wedge e_0.\text{target} = e_1.\text{target} \rightarrow e_0.\text{earliest} \leq e_1.\text{earliest} \end{aligned}$$
**A.12 DBLeaderElection****Event types**

- *DataCommitted*:  $\mathcal{C}$
- *LeaderCommitted*:  $\mathcal{L}$
- *CRead*:  $\mathcal{R}$
- *ECRead*:  $\mathcal{E}$
- *AquireLeader*:  $\mathcal{A}$

**Specification: Unique Leader**

$$\forall e_0, e_1 : \mathcal{A}. e_0.\text{gen} = e_1.\text{gen} \rightarrow e_0.\text{node} = e_1.\text{node}$$
**Specification: Read Consistency**

$$\begin{aligned} \forall e_0 : \mathcal{C}, e_1 : \mathcal{R}. e_1 \prec e_0 \rightarrow e_0.\text{ts} > e_1.\text{ts} \wedge e_0.\text{seq} > e_1.\text{seq} \\ \forall e_0 : \mathcal{C}, e_1 : \mathcal{E}. e_0.\text{seq} \geq e_1.\text{seq} \end{aligned}$$
**Specification: Monotonic Commits**

$$\begin{aligned} \forall e_0, e_1 : \mathcal{C}. e_0 \prec e_1 \rightarrow e_0.\text{gen} \leq e_1.\text{gen} \\ \wedge e_0.\text{ts} < e_1.\text{ts} \wedge e_0.\text{seq} < e_1.\text{seq} \\ \forall e_0, e_1 : \mathcal{L}. e_0 \prec e_1 \rightarrow e_0.\text{gen} \leq e_1.\text{gen} \\ \wedge e_0.\text{ts} < e_1.\text{ts} \wedge e_0.\text{seq} \leq e_1.\text{seq} \end{aligned}$$
**Specification: Atomicity**

$$\begin{aligned} \forall e_0 : \mathcal{T}, e_1 : \mathcal{L}. e_0.\text{status} = \text{COMMITTED} \wedge e_0.\text{gid} = e_1.\text{gid} \rightarrow e_1.\text{status} = \text{COMMITTED} \\ \forall e_0 : \mathcal{U}, e_1 : \mathcal{T}. e_0.\text{gid} = e_1.\text{gid} \wedge e_1.\text{status} = \text{ABORT} \rightarrow e_0.\text{status} = \text{ABORT} \\ \forall e_0 : \mathcal{U} \exists_{|e_0.\text{participants}|} e_1 : \mathcal{S}. e_1 \prec e_0 \wedge e_0.\text{gid} = e_1.\text{gid} \wedge e_1.\text{status} = \text{OK} \\ \forall e_0 : \mathcal{P}, e_1 : \mathcal{L}. e_0.\text{gid} = e_1.\text{gid} \rightarrow e_1 \prec e_0 \wedge e_1.\text{status} = \text{COMMITTED} \wedge e_0.\text{commit\_time} = e_1.\text{commit\_time} \\ \forall e_0 : \mathcal{U}, e_1 : \mathcal{A}. e_0.\text{gid} = e_1.\text{gid} \rightarrow e_0 \prec e_1 \wedge e_0.\text{status} = \text{ABORT} \end{aligned}$$
**Specification: Snapshot Isolation**

$$\begin{aligned} \forall e_0 : \mathcal{R}, e_1 : \mathcal{S}. e_0.\text{gid} = e_1.\text{gid} \rightarrow e_1 \prec e_0 \wedge e_1.\text{status} = \text{OK} \wedge e_0.\text{status} = \text{COMMITTED} \wedge e_0.\text{commit\_time} \geq e_1.\text{prepare\_time} \\ \forall e_0, e_1 : \mathcal{W}. e_1 \prec e_0 \wedge e_0.\text{key} = e_1.\text{key} \rightarrow e_0.\text{commit\_time} \geq e_1.\text{commit\_time} \\ \forall e_0 : \mathcal{R}, e_1 : \mathcal{W}. e_0.\text{start\_time} > e_1.\text{commit\_time} \wedge e_0.\text{key} = e_1.\text{key} \rightarrow e_0.\text{version} \geq e_1.\text{version} \\ \forall e_0 : \mathcal{R}, e_1 : \mathcal{W}. e_0.\text{commit\_time} < e_1.\text{start\_time} \wedge e_0.\text{key} = e_1.\text{key} \rightarrow e_0 \prec e_1 \wedge e_0.\text{version} < e_1.\text{version} \\ \forall e_0 : \mathcal{R} \exists e_1 : \mathcal{W}. e_1 \prec e_0 \wedge e_0.\text{key} = e_1.\text{key} \wedge e_0.\text{val} = e_1.\text{val} \wedge e_0.\text{version} = e_1.\text{version} \end{aligned}$$
**A.13 MVCC-2PC****Event types**

- *eCommitRead*<sup>3</sup>:  $\mathcal{R}$
- *eCommitWrite*<sup>3</sup>:  $\mathcal{W}$
- *eCommitTxn*:  $\mathcal{T}$
- *eRouterStatus*:  $\mathcal{U}$
- *eShardPrep*:  $\mathcal{S}$
- *eLeadCommit*:  $\mathcal{L}$
- *eShardCommit*:  $\mathcal{P}$
- *eShardAbort*:  $\mathcal{A}$



## B Details of PINFER

### B.1 Pseudo-code of generating terms and predicates

---

**Algorithm 1** Bottom-up Term Enumeration

---

**Input** Max. depth  $d$ ; functions  $\mathcal{F}$ ; quantified variables  $\vec{e}$   
**Output** Terms with max depth  $d$ ,  $\mathcal{M}_d$

```

1: procedure ENUM-TERM( $d, \vec{e}, \mathcal{F}$ )
2:   if  $d = 0$  then
3:      $\mathcal{M}_0 := \{x.v_i \mid x \in \vec{e}, v_i \in \text{PAYLOADS-OF}(x)\}$ 
4:      $\mathcal{M}_0 := \mathcal{M}_0 \cup \{\text{indexof}(x) \mid x \in \vec{e}_i\}$ 
5:      $\text{Ord} \leftarrow \{x \mapsto \text{INC-ORDER}() \mid x \in \mathcal{M}_0\}$ 
6:   else
7:     ENUM-TERM( $d - 1, \vec{e}, \mathcal{F}$ )
8:      $\text{Proj} := \{x.v_i \mid x \in \mathcal{M}_{d-1}, v_i \in \text{FIELDS-OF}(x)\}$ 
9:      $\mathcal{M}_d := \{f(\vec{x}) \mid f \in \mathcal{F}, \vec{x} \in \text{ARG}(f, \mathcal{M}_{d-1}, \text{Ord})\}$ 
10:     $\text{Ord} \leftarrow \{x \mapsto \text{INC-ORDER}() \mid x \in \mathcal{M}_d \cup \text{Proj}\}$ 
11:    return  $\text{Proj} \cup \mathcal{M}_d \cup \mathcal{M}_{d-1}$ 
12:   end if
13: end procedure

```

---



---

**Algorithm 2** Construction of Grammar Rules for Predicates

---

**Input**  $\mathcal{S}$  the protocol model;  $\mathcal{F}$  the set of user-provided functions;  $\mathcal{U}$  the set of user-provided predicates;  $\mathcal{M}$  the set of terms;  $\vec{e}$  the event combination  
**Output**  $\mathcal{L}$  production rules of atomic predicates

```

1: procedure CONSTRUCT-GRAMMAR( $\mathcal{S}, \mathcal{P}, \mathcal{M}, \vec{e}$ )
2:    $E, T, P, P_F, P_E \leftarrow \epsilon$ 
3:    $E_i \leftarrow (e_i)_{e_i: \varepsilon_i \in \vec{e}} \quad \triangleright E \text{ Rule}$ 
4:    $\vec{\tau} := \text{ALL-TYPES}(\mathcal{M})$ 
5:    $\text{COMP} := \{<, >, \leq, \geq\}$ 
6:   for  $\vec{x} \in \{\vec{v} \mid \vec{v}_i \in \mathcal{M}, \tau_j \in \vec{\tau}, (\vec{v}_i : \tau_j)\}$  do
7:      $T_{\tau_j} \leftarrow (\vec{x}_0 \mid \vec{x}_1 \mid \dots \mid \vec{x}_n) \quad \triangleright T \text{ Rule}$ 
8:   end for
9:    $T_{\tau_k} \leftarrow (f(\vec{T}_{\kappa_i}))_{(f: \vec{\kappa}_i \rightarrow \tau_k \in \mathcal{F})} \quad \triangleright uf(\vec{T})$ 
10:  for  $\tau_i, \tau_j \in \vec{\tau} \times \vec{\tau}$  do
11:    if  $\tau_i = \tau_j \vee \exists (x : \tau_i = y : \tau_j) \in \mathcal{S}$  then
12:       $P_F \leftarrow (T_{\tau_i} = T_{\tau_j}) \quad \triangleright P_F \text{ Rule}$ 
13:    end if
14:    if  $\exists p(x : \tau_i, y : \tau_j) \in \mathcal{S} \wedge p \in \text{COMP}$  then
15:       $P_F \leftarrow (T_{\tau_i} \text{ OP } T_{\tau_j})_{\text{OP} \in \text{COMP}} \quad \triangleright P_F \text{ Rule}$ 
16:    end if
17:  end for
18:   $P_F \leftarrow (p(\vec{T}_{\kappa_i}))_{(p: \vec{\kappa} \rightarrow \text{bool}) \in \mathcal{U}} \quad \triangleright uP(uf(\vec{T}))$ 
19:   $P_E \leftarrow (E_i \equiv E_j)_{e_i: \varepsilon_i, e_j: \varepsilon_j \in \vec{e} \times \vec{e} \wedge \varepsilon_i = \varepsilon_j \wedge i \neq j} \quad \triangleright P_E \text{ Rule}$ 
20:   $P_E \leftarrow (E_i < E_j \mid P)_{e_i: \varepsilon_i, e_j: \varepsilon_j \in \vec{e} \times \vec{e} \wedge i \neq j} \quad \triangleright P_E \text{ Rule}$ 
21:  return  $(P_F \mid P_E)$ 
22: end procedure

```

---

## C Pruning procedures

PINFER can often learn a large number of target formulas, e.g., it assembled more than 1600 learned formulas for Vertical Paxos after dynamic learning. However, the learned set can contain many *redundant* formulas, i.e., formulas implied by other formulas. Therefore, we implemented logic-based pruning procedures in PINFER to soundly reduce this number by eliminating tautologies, subsumed formulas, and symmetric formulas. These procedures are applied to the learned formulas until reaching a fixpoint, resulting in a set of subsumption-free formulas reported to the user.

**Pruning by syntactic checking.** PINFER abstracts the predicates in  $G$  and  $H$  to propositional variables, and considers  $G$  and  $H$  as sets of propositional variables  $\mathcal{G}$  and  $\mathcal{H}$ , respectively. Then, if  $\mathcal{H} \subseteq \mathcal{G}$ , then  $G \rightarrow H$ . PINFER uses this check to prune tautologies and to detect subsumptions between two formulas with the same quantified events. For instance, given two such formulas

$$I_1 = \forall \vec{e}_i : \vec{\tau}_i. G(\vec{e}_i) \rightarrow \exists \vec{e}_j : \vec{\tau}_j. H(\vec{e}_i, \vec{e}_j)$$

$$I_2 = \forall \vec{e}_i : \vec{\tau}_i. G'(\vec{e}_i) \rightarrow \exists \vec{e}_j : \vec{\tau}_j. H'(\vec{e}_i, \vec{e}_j)$$

If  $G' \rightarrow G$  and  $H \rightarrow H'$ , then  $I_1$  subsumes  $I_2$ . In this case, PINFER keeps  $I_1$  and prunes  $I_2$ . We use syntactic checking first because it is very fast and does not require an SMT solver (e.g., Z3 [12]), although it can miss some semantic implications due to over-approximation in the abstraction of predicates.

**Pruning by semantic checking.** To detect semantic implications, PINFER leverages the Z3 [12] SMT solver to check tautologies and subsumptions. While this can support Basic Predicates ( $P$ ) using their standard interpretations and could treat user-defined predicates/functions as uninterpreted, it may miss an implication that requires additional axioms to be user-provided. By default, when user-defined predicates/-functions appear in the formula, PINFER uses only syntactic checking to guarantee soundness of the pruning procedures.

**Pruning symmetric invariants.** Two formulas are equivalent by symmetry if they are  $\forall$ -only formulas with all quantified variables of the same type, and where one can be obtained from the other by “flipping” arithmetic comparisons. For example, the following two formulas are equivalent by symmetry, and PINFER only keeps one of them:

$$\forall e_0, e_1 : \tau. e_0.x_1 < e_1.x_2 \rightarrow e_0.y_1 < e_1.y_2$$

$$\forall e_0, e_1 : \tau. e_0.x_2 > e_1.x_1 \rightarrow e_0.y_2 > e_1.y_1$$

**Pruning by PChecker.** Although the learned formulas are consistent with the input traces, they may contain false candidates since the traces are likely to under-approximate all behaviors of the protocol. To eliminate false candidates, we generate  $P$  specifications from the learned invariants and check them on a protocol model by utilizing PChecker [13].

Since pruning by PChecker often requires more time than logic-based pruning, this step is optional.