Specy: Automatically Learning Specifications for Distributed Systems from Event Traces

Mike He[†] Ankush Desai[*] Aishwarya Jagarapu[*] Doug Terry
Sharad Malik[†] Aarti Gupta[†]
[†]Princeton University [*]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 developers in practice. Specy addresses this challenge by automatically learning these specifications from *observable event traces* of messages exchanged in distributed systems. Specy uses a specialized grammar for the target specifications based on events, including support for 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 Specy on well-known distributed protocols as well as industrial case studies. Our results demonstrate that for these benchmarks Specy successfully learns the important protocol specifications, including several inductive invariants useful for verification.

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 [14, 44, 58] or state space exploration via model checking [12, 22, 30], to formal proofs using theorem proving [23, 36, 45, 46, 51] or deductive verification [20, 41, 42, 57]. The importance of formal specifications is well-recognized in industry practice [5, 8, 32, 39] 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 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, 12] to reason about the correctness of industry-strength distributed services at AWS, we have consistently observed that the formulation of correctness specifications constitutes the most significant and time-consuming challenge encountered by service teams. Specy aims to mitigate this burden on developers by inferring protocol specifications for distributed systems with minimal user intervention.

1.1 Motivation for our work

Prior research on specification learning can be categorized into two principal approaches. The first encompasses verifier-aided invariant synthesis (e.g., SWISS [18], DuoAI [55]) that generate *inductive invariants* to verify safety properties using deductive verification frameworks [42, 50]. While these approaches are automated and powerful in principle, their practical application is constrained by the requirements for user-provided safety properties, source models expressed in decidable logic fragments, and inherent limitations in verifier performance and scalability. The second category comprises data-driven dynamic learning (e.g., Dinv [16], LIDO [52]) that learn safety properties and invariants by analyzing execution traces of global system states. Although these approaches offer enhanced scalability, they exhibit significant limitations in expressivity – particularly in handling complex quantification patterns such as ∀∃-quantifiers – and frequently necessitate substantial user intervention [16]. Confronted with these limitations, we asked the question: *Can we develop an automated framework for learning specifications of complex protocols, without the constraints imposed by decidable logic models, verifiers, or reduced expressivity?*

This paper presents Specy that answers this question in the affirmative. We identified two additional design principles as requirements satisfied by Specy, for it to be applicable in practice and scalable for real-world distributed systems. First **R1**: Specy learns *protocol specifications* deemed important by system builders in practice, capturing a wide range of protocol specifications for real-world distributed systems (examples are shown in Table 4, §6). Second, **R2**: Specy is agnostic to the system implementation and development languages, and treats the system and its components as black boxes with externally observable behavior that is exhibited in *traces of messages exchanged in a distributed system.* This allows Specy to be applied in many settings where 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.

1.2 Specy: Key Ideas

There are two key insights that enable Specy to address our motivating question and these design requirements.

(1) Learn protocol specifications over messages. The first insight is that most protocol specifications for distributed systems can be formulated over messages using (restricted) first-order logic (FOL). These formulations employ quantifiers ($\forall$, $\forall\exists$) and standard predicates (including equality, numerical comparisons, and temporal relations such as happens-before [29]) 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 [10] 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). In cases where exposing some internal state is required, a user can make relevant state *visible* in event payloads.

(2) Structure of specifications. Our second insight is that a protocol specification over events can be decomposed into the following parts (as marked in Table 1): (1) events over which the specification quantifies, (2) *Guards* G capturing *control conditions*, (3) *Witness* W (optionally) capturing the *existence* of certain events, and (4) *Hypotheses* H stating conditions that must hold under the Guards and the Witnesses. For instance, the *Whitelist Safety* specification of a Firewall protocol [38] in Table 1 is defined over *eRecv* (receiving) and *eGrant* (granting) events. The Guard G states the control condition for allowing the event to be received. The Witness W, $e_1 \prec e_0$ with the traditional happens-before [29] relation ($\prec$), states that when receiving is allowed, then there must *exist* some granting event that happened before. Finally, the Hypotheses H states that an earlier granting event must have granted the permission to the sender. This structure enables Specy to effectively learn specifications by a novel combination of static enumerative search (over events involved in the specifications, G,W) and dynamic learning (to learn H). Furthermore, explicitly identifying Witnesses enables Specy to learn specifications with *quorum constraints* on the number of witnesses (e.g., *Quorum Votes* in Table 1).

1.3 Results and Contributions

We have implemented Specy as a part of the P modeling framework [3, 12], leveraging Daikon [13] 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 Specy, including specifications for Linearizability [21] and Snapshot Isolation [6]. Our evaluations (§6) shows that, *without any user guidance or hints*, specifications learned by Specy can cover 28 of the 43 protocol specifications. The remaining 15 can be learned with light-weight user guidance (discussed in detail in §6). Most of these specifications learned by Specy are beyond the scope of existing approaches. Specy can also learn important specifications that were missed by the developers – these are important in the development process when designing new protocols or refactoring reference implementations for improving performance or adding new features, as these provide insights into the behavior of the system. Additionally, Specy learns inductive invariants of several well-known protocols that were provided manually in a downstream verifier [38].

Contributions. We summarize our contributions as follows:

(1) We present Specy, the first largely automated specification learning framework that learns a *rich class of expressive* protocol specifications from event traces, with support for temporal relations between events, predicates over event payloads, nested $\forall\exists$ quantifiers, and quorum constraints.

(2) The learning procedure in Specy is a novel combination of static enumerative search that allows fine-grained control on the scope of dynamic learning on event traces.

(3) We evaluated Specy on a suite of benchmarks with well-known and proprietary protocols. Our evaluation shows that Specy can learn complex specifications of these protocols, discover new specifications missed by the developers, and find inductive invariants that were manually provided previously for verification by a downstream verifier.

2 Overview of Specy Framework

In this section, we introduce the P language with a running example, provide an overview of the specification formula template used in Specy, and then present the workflow for learning specifications from event traces.

2.1 Background: P Modeling Language

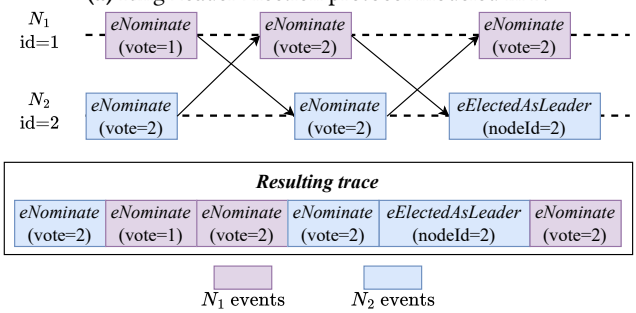
P [3, 12] is a state-machine-based programming language designed for formal modeling and analysis of distributed systems. It enables developers to model system designs as collections of communicating state machines, facilitating checking of both safety and liveness properties. For instance, Amazon Web Services (AWS) used P to validate the strong consistency protocol in Amazon S3 [8], gaining confidence in its correctness during the transition from eventual to

Table 1. Example specifications expressed as formulas over events, with Guards G, Witnesses W, Hypotheses H

Specification Description	Specification Formula
External Consistency (Spanner [10])	$\forall e_0, e_1 : eCommit.$ $e_0.commit < e_1.start$ $\rightarrow$ $e_0.ts < e_1.ts$
Election Safety (Raft [40])	$\forall e_0, e_1 : eElected.$ $e_0.term = e_1.term$ $\rightarrow$ $e_0.leader = e_1.leader$
Update Prop. (Chain Replication [49])	$\forall e_0, e_1 : eNodeLog.$ $e_0.pos \leq e_1.pos$ $\rightarrow$ $e_1.log \preceq_u e_0.log$
Whitelist Safety (Firewall [38])	$\forall e_0 : eRecv.$ $e_0.allowed = \top$ $\rightarrow \exists e_1 : eGrant.$ $e_1 \prec e_0$ $\wedge$ $e_0.src = e_1.host$
Quorum Votes (Consensus [37])	$\forall e_0 : eDecide. \exists_{\geq quorum} e_1 : eVote.$ $e_1 \prec e_0$ $\wedge$ $e_0.ballot = e_1.vote$

```
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 }
```

(a) Ring Leader Election protocol modeled in P.



(b) An example event trace with 2 nodes N_1 andN_2

Figure 1. Ring Leader Election protocol in P

strong consistency. Similarly, DeepSeek employed P to check the correctness of the Fire-Flyer File System (3FS), a high-performance distributed file system [4], showcasing its effectiveness in analyzing complex distributed systems.

2.2 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 [9] 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. 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 the payload.

2.3 Workflow of using Specy in practice

When employing P to verify system design correctness, developers must create both a formal system model and specifications that this model must satisfy. PChecker [3] validates the formal model against these specifications using model-checking-based systematic state space exploration. Creating formal system models is comparatively simpler, as it involves abstracting the system implementation. However, developers struggle with articulating critical specifications that the

system must satisfy and identifying additional properties that provide insights into system behavior.

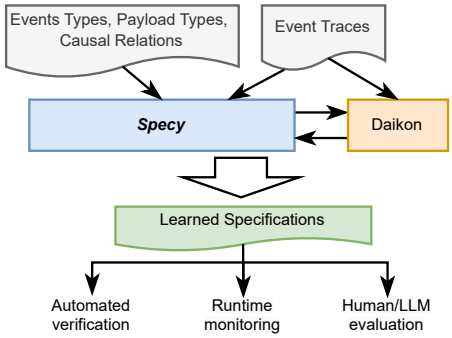


Figure 2. Workflow of Specy

Figure 2 illustrates the Specy workflow: developers first create formal models, then execute PChecker to generate execution traces during exploration, subsequently employ Specy to learn specifications, and finally review these generated specifications to gain a deeper understanding of system behavior. When certain specifications are identified as critical, developers incorporate them into the specification set that must be rigorously tested across both system design and implementation. The specifications generated by Specy serve multiple downstream applications: they enable runtime monitoring of P specifications [8], facilitate deductive verification through PVerifier [3], and provide developers with a comprehensive collection of specifications that characterize system behavior. Furthermore, these specifications can be ranked and interpreted using LLM-based auto-informalization techniques, thereby enhancing developer comprehension of complex system properties.

2.4 Specification Formula Template and Search

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

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

It uses universal quantifiers ($\forall$, at least one) as well as nested 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 sub-formulas shown as Guard ($\boxed{G}$), Witness ($\boxed{W}$), and Hypothesis (H). Note that although this formula template resembles a popular fragment of FOL called EPR [42], Specy does not rely on decidability of the associated logic or availablity of decidable models for learning specifications, unlike many verifier-based approaches [18, 26, 37, 42, 55].

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

Table 2. Example illustrating main steps of Specy

In general, learning specifications poses several challenges, since there is an inherent trade-off between the expressiveness of the learning targets and search efficiency. Specy carefully constrains the search space for learning, and provides an iterative approach to increase the complexity of target specifications. Furthermore, the learning procedure in Specy is fully automated, with the ability to leverage user guidance if required. Specifically, Specy 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. We now introduce a running example for illustrating the important steps in Specy.

2.5 From Event Traces to Specifications

We use the Ring Leader Election protocol to illustrate the important steps in the Specy framework. The well-known safety specification of the protocol, *which Specy does not know in advance*, is that a *unique* leader with the *highest* `id` is *elected*. Specy targets learning the following formulas over the *eElectedAsLeader* ($\mathcal{E}$) and *eNominate* ($\mathcal{N}$) events:

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

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

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

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 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 $e_1 \prec e_0$) [29] 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.

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

We next elaborate on the top-level workflow of Specy (Figure 2) that enables learning these specifications for the Ring Leader Election protocol from event traces.

Inputs to Specy. Specy needs two 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. Specy performs simple static analysis over the P program to extract the latter. It identifies two types of events, *eNominate* ($\mathcal{N}$) and *eElectedAsLeader* ($\mathcal{E}$), where the payloads have fields `vote` and `nodeId`, respectively, of type `tNodeId`, as declared in lines 2 and 3 in Figure 1a. 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 for the Ring Leader Election protocol. 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.

Determining Event Combinations (§4.1). As a first step, Specy heuristically determines events and their respective quantifiers for learning specifications– we call these *event combinations*. Specy automatically identifies a pair with an $\mathcal{E}$ and an $\mathcal{N}$ event, which we use as an illustration, with the results of intermediate steps listed in Table 2; the steps are similar for other event pairs.

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

Learning Procedure (§5). Then, Specy performs the learning procedure, which employs a novel combination of enumerative search (§5.1) and dynamic learning (§5.2). For each event combination, 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), Specy then invokes a dynamic learner (Daikon [13]) that learns *hypotheses* H. Some examples of enumerated G, W, and the corresponding learned H, 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. This is learned because in all traces, the value of `nodeId` in an $\mathcal{E}$ event has appeared as a `vote` from some $\mathcal{N}$ event that occurs before the $\mathcal{E}$ event.

Logic-based pruning procedures (§5.3). The learning procedure can potentially result in a large number of learned specifications, many of which are redundant (e.g., due to subsumption). Specy applies logic-based pruning procedures to eliminate specifications due to tautology and subsumption, resulting in a reduced set to report to the user. In our example, the hypothesis H in Event Combination (a) is learned by Daikon when the enumerated G is $\top$ (since the `nodeId` is higher than the `vote` in *all* $\mathcal{N}$ events). The pruning procedure in Specy identifies this, removes specification (a2), and keeps only specification (a1) with $G = \top$, i.e., Eqn (3).

3 Formula Template: Grammar & Examples

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

3.1 Grammar and Semantics

Specy targets two top-level rules in formula templates, shown as boxed rules in the grammar listed in Figure 3. The first boxed rule, shown as $\phi_{\forall}$, uses only forall ($\forall$) quantifiers over event variables $\vec{e}_i$. The second boxed rule, 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 $\vec{\tau}_i$ and $\vec{\tau}_j$, 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 $\left(\boxed{G(\cdot)}, \boxed{W(\cdot)}, H(\cdot)\right)$ 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 an `indexof` function over event variables. For function applications over terms ($f(\vec{T})$), Specy supports standard arithmetic functions over integers

and floating points, and a `sizeof` function for a list or set that returns its size. The function `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 [29] 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: ≥ 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 ϕ 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 ϕ 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 the grammar. The specification for *External Consistency* of Google Spanner was explained earlier (§1.2). *Election Safety* of the Raft [40] protocol states that a unique leader is elected in each term. *Update Propagation* specification of Chain Replication [49] 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 $\preceq_u$.

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

Causal Relation Patterns		**Event Comb.**	
		$\forall \vec{e_i} : \vec{\tau_i}$	$\exists \vec{e_j} : \vec{\tau_j}$
Same-sourced-sends (sends e_0, sends e_1)		$\langle \tau_0, \tau_1 \rangle$	empty
Receive-then-send (receives e0, then sends e1)		$\langle \tau_0, \tau_1 \rangle$	empty
		$\langle \tau_1 \rangle$	$\langle \tau_0 \rangle$
Send-then-listen (sends e0, then listens to e1)		$\langle \tau_0, \tau_1 \rangle$	empty
		$\langle \tau_0 \rangle$	$\langle \tau_1 \rangle$
Running example	**Lines**		
Same-sourced-sends	21	$\langle \mathcal{E}, \mathcal{E} \rangle$	empty
	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	-	-

The bottom set shows two specifications that require nested existential quantifiers, captured by $\phi_{\forall\exists}$ in the grammar. *Whitelist Safety* of a Firewall protocol [38] was explained in §1.2. *Quorum Votes* of a Consensus protocol [37] states that any decision must have received a quorum of votes, supported by using an augmented existential quantifier with the SC constraint that the number of votes for a decision is at least 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 Specy 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, Specy 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

Specy 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

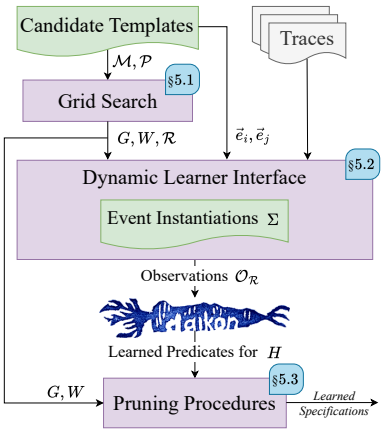


Figure 4. Specy search process: Combining an enumerative Grid Search with dynamic learning by Daikon.

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 table illustrates these patterns on our running example with the listed event combinations on event types, that result from causal patterns identified on the respective lines in the P program (Figure 1a).

Same-sourced-sends. In this pattern, events with types τ_0 and τ_1 (with possibly τ_0 same as τ_1) are sent in the same entry function or event handler. This guides Specy to learn specifications with $\forall$ quantifiers over events with types τ_0 and τ_1.

Receive-then-send. A machine may send out an event of type τ_0 while handling an event of type τ_1, where $\tau_0 \neq \tau_1$. This results in Specy identifying two event combinations, as shown in the table. Notably, the second event combination, where $\vec{e}_j = \langle \tau_0 \rangle$, guides Specy 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 τ_1 can be sent, e.g., after receiving a quorum of events of type τ_0.

Send-then-listen. A machine may send an event of type τ_0 and then start listening for an event of type τ_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 τ_0 sent by a machine, there exists an event of type τ_1 that will be received later by the machine.

4.2 Generating Candidate Templates

Given the event combinations, Specy 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 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, $\mathcal{M} = \{\texttt{indexof}(e_0), \texttt{indexof}(e_1), e_0.nodeId, e_1.vote\}$. Similarly, $\mathcal{P}$ is a set of predicates generated by recursively expanding the "Preds" rule in the grammar (Figure 3), but with restriction 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 (formulas) in Specy combines enumerative search using the Candidate Templates, and dynamic learning from the traces using a dynamic learner such as Daikon [13]. The step-wise workflow is shown in Figure 4. In an outer loop, Specy 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 Specy automatically generates inputs to Daikon in the Dynamic Learner Interface.

5.1 Enumerative Grid Search

To manage the search space complexity of target formulas, Specy 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 $CT = (\mathcal{M}, \mathcal{P})$, Specy enumerates different choices of G and W by conjoining g and w number of predicates, respectively, from $\mathcal{P}$. For learning predicates in H, Specy does not directly control the number, since H is learned by Daikon. Instead, Specy 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 $\mathcal{R}$. Specy depends on the capabilities of the dynamic learner to learn sufficiently expressive predicates in H. By default, Specy 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 (where the w parameter is skipped when learning $\phi_\forall$ formulas).

5.2 Dynamic Learner Interface

Specy provides a Dynamic Learner Interface to prepare inputs to the dynamic learner for learning predicates in H. First, for each event combination $(\vec{e}_i, \vec{e}_j)$, Specy *filters* given trace $(\mathcal{T} \in \mathbb{T})$ by removing events of types that do not match $\vec{e}_i$ or $\vec{e}_j$, preserving 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), SPECY computes *instantiations*, denoted σ, 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 σ, we can compute an *evaluation* of a term T or a predicate P, denoted $[\![T]\!]_\sigma$, $[\![P]\!]_\sigma$, respectively, by instantiating symbolic events in T, P with the corresponding concrete events in σ. In particular, an evaluation under σ results in a concrete value (e.g., an integer) for a term, and a true ($\top$) or false ($\bot$) value for a predicate. We extend evaluations to conjunctions of predicates in G, W, denoted as $[\![G]\!]_\sigma$, $[\![W]\!]_\sigma$, respectively, which result in true/false values.

Learning $\phi_\forall$ formulas. SPECY computes a set of instantiations, denoted Σ, as follows:

$$\Sigma = \bigcup_{traces} \{\sigma \mid [\![G]\!]_\sigma = \top\}$$

Intuitively, Σ represents all instantiations σ of symbolic events by concrete events in the given traces, such that G evaluates to true under σ. In this case, a predicate P_H, such that $[\![P_H]\!]_\sigma$ 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, SPECY computes a set of observations resulting from evaluations of terms in $\mathcal{R}$ under each $\sigma \in \Sigma$, denoted $O_\mathcal{R}$, as input to the dynamic learner. Recall that in the grid search, SPECY 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 SPECY computed the following observations $O_\mathcal{R}$ (where $t \mapsto v$ indicates that term t evaluates to value v under some σ):

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

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 $O_\mathcal{R}$.

Learning $\phi_{\forall\exists}$ formulas. For handling the existential quantifiers, SPECY computes the set Σ as a set of tuples:

$$\Sigma = \bigcup_{traces} \{(\sigma_i, \omega(\sigma_i)) \mid [\![G]\!]_{\sigma_i} = \top \wedge \forall \sigma_j \in \omega(\sigma_i).\, [\![W]\!]_{\sigma_i \cup \sigma_j} = \top\}\}$$

In each tuple, the first component is an instantiation σ_i for symbolic events $\vec{e}_i$ such that G holds on the concrete events (similar to the $\phi_\forall$ 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 σ_i and σ_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 σ_j for some σ_i, i.e., $\omega(\sigma_i)$ is empty, then the associated W and Σ are discarded, and SPECY proceeds to the next W in the grid search.

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

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

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

Now, SPECY leverages a dynamic learner to find predicates H that hold over (O_i, O_{jk}) for *some* O_{jk} in each tuple in $O_\mathcal{R}$ (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 $\vec{e}_j = \langle e_1 : \mathcal{N} \rangle$, SPECY has enumerated $G = \top$, $W = e_1 \prec e_0$, 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 SPECY computes $O_\mathcal{R}$ for some given trace as shown below (with only one tuple shown for ease of exposition):

$$O_\mathcal{R} = \left\{\left(\begin{array}{l}\{e_0.nodeId \mapsto 4\},\\ \{\{e_1.vote \mapsto 2\}, \{e_1.vote \mapsto 4\}, \{e_1.vote \mapsto 4\}\}\end{array}\right)\right\}$$

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

After Daikon discovers the predicate $e_0.ballot = e_1.vote$ in H, SPECY 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): $\forall e_0 : \mathcal{E}.\, \exists e_1 : \mathcal{N}.\, e_1 \prec e_0 \wedge e_0.nodeId = e_1.vote$.

Learning Set Cardinality constraints. Recall that a Set Cardinality (SC) constraint can be used to augment existential quantification in $\phi_{\forall\exists}$ formulas (§3.1). SPECY can also learn an SC constraint by using Daikon. We illustrate this process on the *Quorum Votes* specification of the Consensus protocol [37] shown in Table 1, stateing that *every decision (eDecide) receives a majority of votes (eVote)*. 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 Specy to consider traces that are generated from systems with different number of nodes, i.e., with different constants for the required quorum size, and each trace has an event with the required *quorum*.

Now, when Specy performs grid search for the event combination $\vec{e}_i = \langle e_0 : eDecide \rangle$, $\vec{e}_j = \langle e_1 : eVote \rangle$, it enumerates $G = \top$ and $W = (e_1 \prec e_0) \wedge (e_0.ballot = e_1.vote)$. According to the protocol semantics, an *eDecide* is emitted only after a majority of *eVote* events satisfying W have been sent. Therefore, for every σ_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 $[\![W]\!]_{\sigma_i \cup \sigma_j}$ is true for all $\sigma_j \in \omega(\sigma_i)$ and $|\omega(\sigma_i)| \geq quorum$. Thus, Daikon *automatically learns that the predicate* $|\omega(\sigma_i)| \geq quorum$ holds over all such witness events. Specy 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* (UG3, §3.3). Specy automatically learns the shown specification, including the SC constraint on the augmented existential quantifier.

5.3 Formula Assembly and Pruning Procedures

After the dynamic learner has learned predicates for H, Specy 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, Specy 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. Specy 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 Specy 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 (detailed in Appendix B.2).

Pruning by syntactic checking. Specy abstracts the predicates in G, H to propositional variables $\mathcal{G}, \mathcal{H}$, respectively. Then, if $\mathcal{H} \subseteq \mathcal{G}$, then $G \rightarrow H$. Specy uses this implication 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, Specy uses the Z3 solver [11] (user-defined predicates considered uninterpreted).

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. Specy can (optionally) generate P specifications from the learned formulas and check them on the P program via PChecker [12], to eliminate formulas that are reported false.

6 Evaluations

Our evaluation addresses the following research questions:
RQ1: Can Specy effectively learn protocol specifications that developers consider essential correctness properties?
RQ2: How does Specy benefit development in practice?
RQ3: Do the specifications learned by Specy help deductive verification, particularly for learning inductive invariants?
RQ4: How does Specy compare to existing approaches? We evaluated Specy against state-of-the-art tools SWISS [18] and DuoAI [55] on a complex benchmark for comparison.

6.1 Implementation

We implemented Specy as an extension of the P compiler [3, 12], 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 [13]. Specy uses the Z3 solver [11] for semantic checking in pruning. As a specialized mode within P, Specy accepts formal models from P users, autonomously executes the model checker to generate event traces, and subsequently presents the learned specifications to the user.

6.2 Benchmarks

We evaluated Specy 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 $\top$ in the "Benchmark" column of Table 4) from Ivy to P that were considered as benchmarks in previous papers, namely, IC3FOL [2, 26], SWISS [18], and I4 [1, 37], and a Firewall model from PVerifier [3, 38]. In addition, we developed P models for 6 foundational protocols: Two-Phase Commit (2PC) [17], Ring Leader Election [9], Paxos [31], Chain Replication [49], Vertical Paxos [33] and Raft [40]. Notably, our Raft model is fully functional (excluding cluster reconfiguration), with two failure models: unreliable network communications and node crashes.

Real-world case studies. To demonstrate the applicability of Specy on real-world case studies, we considered three protocol models that were developed by service teams at AWS. 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

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 Specy. The next column provides a description (and if ∀∃ is needed). The last column shows the end-to-end run time of Specy (in seconds).

Benchmark	LoC	S_{goals}	Description of Specifications	Time (s)
Ring Leader Election [9]	47	3 ✓	Unique leader, Leader Highest ID, Leader Voted	352
Consensus$^{\top}$[26]	61	1 ✓	Safety (Unique Decision)	925
2PC [3]	238	2 ✓	Atomicity ($\forall\exists_n$), Safety (Commit or Abort)	3935
Sharded KV$^{\top *}$[26]	48	1 ✓	Safety (Unique Shard Owner)	481
Paxos [31]	164	2 ✓	Safety (Unique Decision), B3($\mathcal{B}$)†	1155
Distributed Lock$^{\top}$[37]	73	1 ✓	Safety (Mutual Exclusion)	902
Vertical Paxos [33]	241	2 ✓	Safety (Unique Decision), B3($\mathcal{B}$)†	2589
Firewall [38]	80	1 ✓	Whitelisted Safety (∀∃)	753
Lock Server$^{\top}$[37]	102	1 ✓	Safety (Mutual Exclusion)	620
Chain Replication [49]	452	6 ✓	Update Propagation^{+},Inprocess Requests^{+},Linearizability (∀∃)$^{\dagger\#}$	984
Raft*[40]	1191	5 ✓	Election Safety, State Machine Safety, Log Matching^{+} Leader Append-Only, Leader Completeness^{+}	4223
GlobalClock	177	3 ✓	Clock Monotonicity, Real-time ordering$^{\#}$	805
DBLeaderElection	1178	5 ✓	Monotonic Commits, Unique Leader, Read Consistency$^{\dagger\#}$	697
MVCC-2PC	1135	10 ✓	Atomicity ($\forall\exists_n$),$^{\#}$Snapshot Isolation (∀∃)$^{\ddagger\#}$	6642

$^{\top}$ P model translated from an IVy model
* Sharded KV models key transfers but not modifications, Raft omits cluster reconfiguration
† Requires user-specified event combinations (UG1)
$^{+}$ Requires user-defined predicate over log entries exposed to traces via events (UG2 & UG3)
‡ Requires events that expose commit logs to traces $^{\#}$ Implied by a subset of the learned specifications

transactional service that combines multi-version concurrency control [6] with two-phase commit [17] to guarantee atomicity and snapshot isolation. The formal models and specifications were written by the developers, we demonstrate that Specy can automatically learn specification that the teams had written when building these services. More importantly, Specy also learned important specifications that were missing in these handwritten specifications.

Setup. The experiments were conducted on an AMD EPYC 7R13 192-core CPU with 1.5TB of memory. For each benchmark, input traces are generated using P model checker [12], 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 and 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

The minimum requirement for Specy to be practical as a specification learning framework is being able to learn known specifications of existing protocols. To evaluate whether Specy satisfies this requirement, 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 [9, 26, 31, 33, 37, 40, 42, 49], and for industrial case studies, specifications authored by the developers themselves. We designate these specifications as *goal specifications* that should be learned by Specy. The complete set of first-order logic (FOL) formulas constituting these specifications is presented in Appendix A. Notably, 5 of these protocol specifications require ∀∃-quantifiers, including 2 with quorum constraints. 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 Specy. Footnotes in the table denote instances where user guidance (UG) was necessary to successfully learn the goal

Table 5. Number of specifications remaining after each pruning step. S_{raw} and S_{sem} columns show the number of specifications before and after all the pruning procedures, where the reduction ratio ($S_{\text{raw}}/S_{\text{sem}}$) is shown in *RR* column.

Benchmark	S_{raw}	S_{syn}	S_{sem}	*RR*	S_{false}
Ring Leader Elect.	479	52	30	15×	0
Consensus	361	28	28	13×	1
2PC	887	60	46	19×	9
Sharded KV	289	23	19	15×	0
Paxos	850	52	49	17×	5
Distributed Lock	740	91	77	9×	0
Vertical Paxos	1650	92	76	21×	5
Firewall	606	46	40	19×	4
Lock Server	399	44	35	11×	0
Chain Replication	533	40	37	14×	2
Raft	1080	73	58	18×	10
GlobalClock	368	37	37	9×	0
DBLeaderElect.	577	50	47	12×	9
MVCC-2PC	5789	334	236	24×	0

specifications. *Notably, all 43 are covered by the protocol specifications learned by* Specy *for these benchmarks*, 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 Specy and user guidance that proved essential for learning protocol specifications.

When is user guidance needed? Specy 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 Specy'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 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.

6.4 RQ2: Benefits of Specy in practice

Specy benefits the development by presenting a human-tractable set of protocol specifications, which can provide insights into system behaviors for the developers.

Human-tractable learned specifications. We evaluate the effectiveness of the pruning procedures, showing that Specy can reduce the learned specifications to a reasonable number. Table 5 shows the number of specifications remaining after each pruning procedure. The dynamic learner Daikon [13] can learn a large number of specifications (shown in the S_{raw} column). Specy first applies sanitization and syntactic-based pruning that prunes vacuous and redundant specifications, resulting in a significantly smaller set (shown in the S_{syn} column). Then, to detect redundancies that are missed by syntactic checking, Specy applies pruning by semantic checking (shown in the S_{sem} column), and presents the output to the user. The last two columns report the Reduction Ratio (RR = $S_{\text{raw}}/S_{\text{sem}}$) and the number of specifications falsified with PChecker (S_{false}), i.e., PChecker found execution traces where these specifications evaluate to false.

Note that 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, with a geometric average reduction ratio of 14.8×. Importantly, these procedures did not eliminate any specifications Specy covered in RQ1. We have validated using the PChecker [3] that all the remaining specifications in the S_{sem} are correct.

Specifications overlooked by developers. Specifications learned by Specy provides insights into the systems behavior and also highlight some missing specifications that are crucial for system correctness. For example, Specy learned two critical specifications that were overlooked by the developers in the proprietary MVCC-2PC protocol (with a sharded storage as the application), identified in Specy's output by team members familiar with the protocol: (1) *only the leader shard server can initiate a commit of transaction*, and (2) *each transaction can only be prepared at most once by each shard server*. If violated, any shard server may initiate a commit and prepare multiple times by itself, resulting in a committed transaction without consensus. These were not present in the hand-written specifications. In general, *Specy can benefit developers by identifying specifications that are overlooked when designing systems.*

6.5 RQ3: Learning inductive invariants

To evaluate Specy's utility for downstream verification tasks, we conducted experiments with benchmarks previously verified in PVerifier [38]—a framework for deductive verification of P programs. Our evaluation focused specifically on the *inductive invariants* required by PVerifier proofs that were

Table 6. Results for learning inductive invariants using Specy (RQ3). We use a subset of benchmarks with proofs in PVerifer [38]. 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*	1	2	✓
Consensus*	1	5	✓
Distributed Lock*	4	4	✓
Lock Server*	4	4	✓
Firewall	1	0	✓
Sharded KV*	1	1	✓

previously provided manually by users (formulas are provided in Appendix A). These inductive invariants fall into two categories: event-based and state-based, with the latter requiring additional user-defined instrumentation (for adding events that expose host 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 $*$), Specy successfully learns all inductive invariants. Without such instrumentation, Specy still learns all event-based invariants (I_e) but fails to learn the state-based ones (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 Specy with SWISS [18] and DuoAI [55] on Vertical Paxos, the most computationally demanding open-source benchmark for Specy. Using a 24-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 (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 upto eight literals – while its verifier struggles with complex queries. In contrast, Specy learned both safety correctness specifications and additional (likely) invariants for Vertical Paxos in 6 hours.

7 Related Work

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

Verifier-aided invariant synthesis. SWISS [18], DuoAI [55] are state-of-the-art tools for learning inductive invariants (with ∀∃-quantifiers) to formally prove safety properties using verifiers [42, 50]. These approaches face applicability and scalability limitations as discussed in §1.1. 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 [15, 25, 26, 37, 43, 56] have similar limitations, and some do not support nested existential quantifiers.

Data-driven invariant learning. Dinv [16], LIDO [52] 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 on supporting more expressive specifications with ∀∃-quantifiers and requiring user intervention for manual refinement on the outputs as discussed in §1.1.

Other earlier efforts [7, 27, 53, 54] focused on learning *property patterns* over execution traces or message sequence charts. Specifically, Perracotta [54] uses pre-defined property templates to recognize patterns, Beschastnikh et al. [7] is tailored for finding temporal relationships between events, and Kumar et al. [27] 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 Specy.

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 [36, 45, 46, 51], using interactive theorem-provers such as Rocq and Iris [24, 48], often require high manual effort to formulate and find correctness proofs. Semi-automated efforts [19, 20, 34, 38, 41, 47, 57] leverage user-provided annotations to formulate verification tasks handled by automated verifiers such as Ivy [42, 50], Dafny [35], or Uclid [28]. Among these, a recent work Kondo [57] 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 Specy, 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 [12, 22, 30], although their scalability is limited to small systems. Specy can utilize traces generated by these techniques, and also use them to potentially falsify the learned specifications.

8 Conclusion

We present Specy, 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 ∀∃-quantifiers. Specy 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 Specy can learn all the protocol specifications from our benchmark suite of 11 well-known distributed protocols and 3 proprietary industrial-scale protocols, including specifications that were missing for a proprietary protocol. We additionally show that Specy 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] 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.

[5] Robert Beers. Pre-rtl formal verification: An intel experience. In *2008 45th ACM/IEEE Design Automation Conference*, pages 806–811, 2008.

[6] 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.

[7] 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.

[8] Marc Brooker and Ankush Desai. Systems correctness practices at aws: Leveraging formal and semi-formal methods. *Queue*, 22(6):79–96, February 2025.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Jim Gray. *The transaction concept: virtues and limitations*, page 140–150. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[30] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[31] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[32] Leslie Lamport. Industrial Use of TLA+ — lamport.azurewebsites.net. https://lamport.azurewebsites.net/tla/industrial-use.html, 2019. [Accessed 10-03-2025].

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. Message chains for distributed system verification. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[46] 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.

[47] Nikhil Swamy, Cătălin Hriţcu, 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.

[48] The Coq Development Team. The coq proof assistant – https://doi.org/10.5281/zenodo.14542673, September 2024.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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. Events that are marked with * means that they are added via user guidance UG3. English descriptions of each specification are placed below the formula.

A.1 Ring Leader Election protocol

Event types
eElectedAsLeader: $\mathcal{E}$ *eNominate*: $\mathcal{N}$

Specification: Unique Leader

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

This specification states that the elected leader is unique, under the setting where *nodeId* assigned to each node are distinct. In our setting where the `Node` machine immediate transitions into `Won` state after receiving its own `nodeId` as a `vote`, this specification alone is bogus since there can be only 1 $\mathcal{E}$ event in a single round of leader election. Therefore, we include the following two for strengthening:

Specification: Leader highest

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

The leader has the highest `nodeId`.

Specification: Leader elected

$\forall e_0 : \mathcal{E}\ \exists e_1 : \mathcal{N}.\ e_1 \prec e_0 \wedge e_1.nodeId = e_1.vote$

The leader must have been voted in an earlier $\mathcal{N}$ event.

Inductive Invariants

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

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

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

We applied user guidance to include the `nxt` field, which is the `nodeId` of the node to the right, resulting in the three inductive invariants. The first states that vote sent to the same target is always monotonically increasing. The second states that the `nodeId` of the $\mathcal{E}$ event is the highest

A.2 2PC

Event types
eCommitTxn: $\mathcal{C}$ *eAbortTxn*: $\mathcal{A}$ *ePrepareSuccess*: $\mathcal{P}$

Specification: Atomicity

$\forall e_0 : \mathcal{C}\ \exists_n e_1 : \mathcal{P}.\ e_1 \prec e_0 \wedge e_0.txnId = e_1.txnId$

If a transaction is committed ($\mathcal{C}$), then there must exists n, which is the number of participants, successful prepares ($\mathcal{P}$) that happened before.

Specification: Commit or Abort

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

A transaction cannot be both committed and aborted.

A.3 Sharded KV

Event types
eOwns*: $\mathcal{O}$ *eTransfer*: $\mathcal{T}$

Specification: Unique Shard Owner

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

Each key is owned by a unique shard.

Inductive Invariants

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

$\forall e_0, e_1 : \mathcal{T}.\ e_0.key = e_1.key \rightarrow$
$\quad e_0.dst = e_1.dst \wedge e_0.value = e_1.value$

A.4 Paxos

Event types
eLearn: $\mathcal{D}$ *eAcceptReq*: $\mathcal{A}$

Specification: Unique Decision

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

Decisions carry a unique decision value.

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

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

If a decision has been made in a smaller round, then the higher-numbered round should propose the decided value in the smaller round.

A.5 Distributed Lock

Event types
eHasLock: $\mathcal{L}$ *eNodeState**: $\mathcal{N}$ *eTransfer*: $\mathcal{T}$

Specification: Mutual Exclusion

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

The node that acquired the lock is unique in a single epoch of lock acquiring.

Inductive Invariants

$\forall e_0, e_1 : \mathcal{N}.\ e_0.hasLock \land e_0.epoch = e_1.epoch \rightarrow$
$\quad e_0.node = e_1.node$
$\forall e_0, e_1 : \mathcal{T}.\ e_0.round = e_1.round \rightarrow$
$\quad e_0.node = e_1.node \land e_0.epoch = e_1.epoch$
$\forall e_0 : \mathcal{N}.\ e_0.hasLock \rightarrow \exists e_1 : \mathcal{T}.\ e_1 \prec e_0 \land$
$\quad e_0.node = e_1.node \land e_0.epoch = e_1.epoch \land$
$\quad e_0.round = e_1.round$
$\forall e_0 : \mathcal{L}, e_1 : \mathcal{N}.\ e_0.round = e_1.round \land \neg e_1.hasLock \rightarrow$
$\quad e_0.epoch > e_1.epoch$
$\forall e_0 : \mathcal{L}, e_1 : \mathcal{N}.\ e_0.round = e_1.round \rightarrow$
$\quad e_0.epoch \geq e_1.epoch$
$\forall e_0 : \mathcal{N}, e_1 : \mathcal{T}.\ e_1 \prec e_0 \land e_0.hasLock \rightarrow$
$\quad e_0.epoch \geq e_1.epoch$
$\forall e_0 : \mathcal{N}, e_1 : \mathcal{T}.\ e_0 \prec e_1 \rightarrow e_0.epoch < e_1.epoch$
$\forall e_0, e_1 : \mathcal{T}.\ e_0 \prec e_1 \rightarrow e_1.epoch < e_1.epoch$

A.6 Vertical Paxos

Event types
eDecided: $\mathcal{D}$ *ePropose*: $\mathcal{P}$

Specification: Unique Decision

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

The decision value is unique in decision events.

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

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

If a value has been decided in an earlier configuration, proposers from the current configuration can only propose the decided value from the earlier configuration.

A.7 Firewall

Event types *eRecv*: $\mathcal{S}$ *eGrant*: $\mathcal{A}$ *SentFromInternal*: $\mathcal{I}$

Specification: Whitelist Safety

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

If the receiving is allowed, then there exist some granting event that grant the permission to the sender.

Inductive Invariant

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

A.8 Lock Server

Event types
eHoldsLock: $\mathcal{L}$ *eServerState**: $\mathcal{S}$
eGrant: $\mathcal{G}$ *eUnlock*: $\mathcal{U}$

Specification: Mutual Exclusion

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

The node that acquires the lock is unique in each epoch.

Inductive Invariants

$\forall e_0, e_1 : \mathcal{G}.\ e_0.epoch = e_1.epoch \rightarrow e_0.node = e_1.node$
$\forall e_0, e_1 : \mathcal{U}.\ e_0.epoch = e_1.epoch \rightarrow e_0.node = e_1.node$
$\forall e_0 : \mathcal{L}, e_1 : \mathcal{G}.\ e_0.epoch = e_1.epoch \rightarrow e_1 \prec e_0$
$\forall e_0 : \mathcal{S}, e_1 : \mathcal{G}.\ e_0.epoch = e_1.epoch \land e_1 \prec e_0 \rightarrow$
$\quad \neg e_0.holdsLock$
$\forall e_0 : \mathcal{S}, e_1 : \mathcal{U}.\ e_0.epoch = e_1.epoch \land e_1 \prec e_0 \rightarrow$
$\quad \neg e_0.holdsLock$
$\forall e_0 : \mathcal{S}, e_1 : \mathcal{L}.\ e_0.epoch = e_1.epoch \land e_1 \prec e_0 \rightarrow$
$\quad \neg e_0.holdsLock$
$\forall e_0 : \mathcal{L}, e_1 : \mathcal{U}.\ e_0.epoch = e_1.epoch \rightarrow e_0 \prec e_1$
$\forall e_0 : \mathcal{G}, e_1 : \mathcal{U}.\ e_0.epoch = e_1.epoch \rightarrow e_0 \prec e_1$

A.9 ChainReplication

Event types
eReadSuccess: $\mathcal{R}$ *eWriteResponse*: $\mathcal{W}$ *eNotifyLog**: $\mathcal{N}$

Specification: Update Propagation

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

If a node n_0 is placed before a node n_1 in the chain, then the log of n_1 is a prefix of the log of node n_0. 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

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

If a node n_0 is placed before a node n_1, then the log of n_0 equals the *sent but not acked* log union with the log of n_1, ordered by time of occurrence. 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

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

The first states that the version number in the read response for the same key increases monotonically;
the second states that if a read to a key k happens after the write to k, then the version in the read response is at least as new as the version of the write response.
the third formula states that if some value is successfully read, then there exists a previous write to that key;
the third formula states that writes to the same key always returns the latest version.
Together, these formulas state that a read always reads the latest write.

A.10 Raft

Event types
eBecomeLeader: $\mathcal{B}$ *eEntryApplied*: $\mathcal{A}$ *eNotifyLog**: $\mathcal{N}$

Specification: Election Safety

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

A unique leader is elected in each term

Specification: State Machine Safety

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

If the applied entry is on the same index, then they have the same payload.

Specification: Leader Append-Only

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

The leader only appends to the log but does not remove from it

---

Specification: Leader Completeness

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

A leader in a later term contains all the previous applied logs

---

Specification: Log Matching

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

If two logs l_0 and l_1 on two different nodes agree on an index i, then l_0 and l_1 also agree on all preceding indices $j < i$.

A.11 GlobalClock

Event types
eLocalClock: $\mathcal{L}$

Specification: Clock Monotonicity

$\forall e_0, e_1 : \mathcal{L}.\ e_0.latest < e_1.latest \rightarrow$
$e_0.trueTime < e_1.trueTime$

The latest bound of the clock respects the true time.

Specification: Real-time ordering

$\forall e_0, e_1 : \mathcal{L}.\ e_0.trueTime < e_1.trueTime \rightarrow$
$e_0.earliest < e_1.latest$

$\forall e_0, e_1 : \mathcal{L}.\ e_0.trueTime < e_1.trueTime$
$\wedge\ e_0.target = e_1.target \rightarrow$
$e_0.earliest \leq e_1.earliest$

The time bounds reflect the true time. This is similar to Google Spanner's External Consistency [10].

A.12 DBLeaderElection

Event types
DataCommited: $\mathcal{C}$ *LeaderCommited*: $\mathcal{L}$ *CRead*: $\mathcal{R}$
ECRead: $\mathcal{E}$ *AquireLeader*: $\mathcal{A}$

Specification: Unique Leader

$\forall e_0, e_1 : \mathcal{A}.\ e_0.gen = e_1.gen \rightarrow e_0.node = e_1.node$

A unique leader is elected for each generation.

Specification: Read Consistency

$\forall e_0 : \mathcal{C}, e_1 : \mathcal{R}.\ e_1 \prec e_0 \rightarrow e_0.ts > e_1.ts \wedge e_0.seq > e_1.seq$

$\forall e_0 : \mathcal{C}, e_1 : \mathcal{E}.\ e_0 \prec e_1 \rightarrow e_0.seq \geq e_1.seq$

Specification: Monotonic Commits

$\forall e_0, e_1 : \mathcal{C}.\ e_0 \prec e_1 \rightarrow e_0.gen \leq e_1.gen$
$\wedge\ e_0.ts < e_1.ts \wedge e_0.seq < e_1.seq$

$\forall e_0, e_1 : \mathcal{L}.\ e_0 \prec e_1 \rightarrow e_0.gen \leq e_1.gen$
$\wedge\ e_0.ts < e_1.ts \wedge e_0.seq \leq e_1.seq$

If a commit C_0 happens before a commit C_1, then C_1 commits the more updated data than C_0.

A.13 MVCC-2PC

Event types
eCommitRead*: $\mathcal{R}$ *eCommitWrite**: $\mathcal{W}$ *eCommitTxn*: $\mathcal{T}$
eRouterStatus: $\mathcal{U}$ *eShardPrep*: $\mathcal{S}$ *eLeadCommit*: $\mathcal{L}$
eShardCommit: $\mathcal{P}$ *eShardAbort*: $\mathcal{A}$

Specification: Atomicity

$\forall e_0 : \mathcal{T}, e_1 : \mathcal{L}.\ e_0.status = \texttt{COMMITTED} \wedge e_0.gid = e_1.gid \rightarrow$
$e_1.status = \texttt{COMMITTED}$

$\forall e_0 : \mathcal{U}, e_1 : \mathcal{T}.\ e_0.gid = e_1.gid \wedge e_1.status = \texttt{ABORT} \rightarrow$
$e_0.status = \texttt{ABORT}$

$\forall e_0 : \mathcal{U}\ \exists_{|e_0.participants|} e_1 : \mathcal{S}.\ e_1 \prec e_0 \wedge$
$e_0.gid = e_1.gid \wedge e_1.status = \texttt{OK}$

$\forall e_0 : \mathcal{P}, e_1 : \mathcal{L}.\ e_0.gid = e_1.gid \rightarrow e_1 \prec e_0 \wedge$
$e_1.status = \texttt{COMMITTED} \wedge$
$e_0.commit\_time = e_1.commit\_time$

$\forall e_0 : \mathcal{U}, e_1 : \mathcal{A}.\ e_0.gid = e_1.gid \rightarrow e_0 \prec e_1 \wedge$
$e_0.status = \texttt{ABORT}$

Specification: Snapshot Isolation

$\forall e_0 : \mathcal{R}, e_1 : \mathcal{S}.\ e_0.gid = e_1.gid \rightarrow e_1 \prec e_0 \wedge$
$e_1.status = \texttt{OK} \wedge e_0.status = \texttt{COMMITTED} \wedge$
$e_0.commit\_time \geq e_1.prepare\_time$

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

$\forall e_0 : \mathcal{R}, e_1 : \mathcal{W}.\ e_0.start\_time > e_1.commit\_time \wedge$
$e_0.key = e_1.key \rightarrow e_0.version \geq e_1.version$

$\forall e_0 : \mathcal{R}, e_1 : \mathcal{W}.\ e_0.commit\_time < e_1.start\_time \wedge$
$e_0.key = e_1.key \rightarrow e_0 \prec e_1 \wedge$
$e_0.version < e_1.version$

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

B Details of Specy

B.1 Pseudo-code for 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, $\mathcal{M}_d$

procedure Enum-Term($d, \vec{e}, \mathcal{F}$)
  if $d = 0$ **then**
    $\mathcal{M}_0 \coloneqq \{x.v_i \mid x \in \vec{e}, v_i \in \textsc{Payloads-Of}(x)\}$
    $\mathcal{M}_0 \coloneqq \mathcal{M}_0 \cup \{\texttt{indexof}(x) \mid x \in \vec{e}_i\}$
    $Ord \leftarrow \{x \mapsto \textsc{Inc-Order}() \mid x \in \mathcal{M}_0\}$
  else
    Enum-Term($d-1, \vec{e}, \mathcal{F}$)
    Proj $\coloneqq \{x.v_i \mid x \in \mathcal{M}_{d-1}, v_i \in \textsc{Fields-Of}(x)\}$
    $\mathcal{M}_d \coloneqq \{f(\vec{x}) \mid f \in \mathcal{F}, \vec{x} \in \textsc{Arg}(f, \mathcal{M}_{d-1}, Ord)\}$
    $Ord \leftarrow \{x \mapsto \textsc{Inc-Order}() \mid x \in \mathcal{M}_d \cup \text{Proj}\}$
    return Proj $\cup \mathcal{M}_d \cup \mathcal{M}_{d-1}$
  end if
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

procedure Construct-Grammar($\mathcal{S}, \mathcal{P}, \mathcal{M}, \vec{e}$)
  $E, T, P, P_F, P_E = \epsilon$
  $E_i \leftarrow (e_i)_{e_i:\varepsilon_i \in \vec{e}}$ ▷ E Rule
  $\vec{\tau} \coloneqq \textsc{All-Types}(\mathcal{M})$
  $\textsc{Comp} \coloneqq \{<, >, \leq, \geq\}$
  for $\vec{x} \in \{\vec{v} \mid \vec{v}_i \in \mathcal{M}, \tau_j \in \vec{\tau}, (\vec{v}_i : \tau_j)\}$ **do**
    $T_{\tau_j} \leftarrow (\vec{x}_0 \mid \vec{x}_1 \mid \cdots \mid \vec{x}_n)$ ▷ T Rule
  end for
  $T_{\tau_k} \leftarrow \left(f(\vec{T}_{\kappa_i})\right)_{(f:\vec{\kappa}_i \to \tau_k \in \mathcal{F})}$ ▷ $uf(\vec{T})$
  for $\tau_i, \tau_j \in \vec{\tau} \times \vec{\tau}$ **do**
    if $\tau_i = \tau_j \vee \exists (x : \tau_i = y : \tau_j) \in \mathcal{S}$ **then**
      $P_F \leftarrow \left(T_{\tau_i} = T_{\tau_j}\right)$ ▷ P_F Rule
    end if
    if $\exists p(x : \tau_i, y : \tau_j) \in \mathcal{S} \wedge p \in \textsc{Comp}$ **then**
      $P_F \leftarrow \left(T_{\tau_i} \text{ op } T_{\tau_j}\right)_{\text{op} \in \textsc{Comp}}$ ▷ P_F Rule
    end if
  end for
  $P_F \leftarrow \left(p(\vec{T}_{\kappa_i})\right)_{(p:\vec{\kappa} \to bool) \in \mathcal{U}}$ ▷ $uP(uf(\vec{T}))$
  $P_E \leftarrow \left(E_i \equiv E_j\right)_{e_i:\varepsilon_i, e_j:\varepsilon_j \in \vec{e} \times \vec{e} \wedge \varepsilon_i = \varepsilon_j \wedge i \neq j}$ ▷ P_E Rule
  $P_E \leftarrow \left(E_i \prec E_j \mid P\right)_{e_i:\varepsilon_i, e_j:\varepsilon_j \in \vec{e} \times \vec{e} \wedge i \neq j}$ ▷ P_E Rule
  return $(P_F \mid P_E)$
end procedure

B.2 Pruning Procedures

Specy 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 Specy 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. Specy 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 $G \to H$. Specy uses this check to prune tautologies and to detect subsumptions between two formulas with the same quantified events. For instance, given two such formulas

$$S_1 = \forall \overrightarrow{e_i : \tau_i}.\, G(\vec{e}_i) \to \exists \overrightarrow{e_j : \tau_j}.\, H(\vec{e}_i, \vec{e}_j)$$
$$S_2 = \forall \overrightarrow{e_i : \tau_i}.\, G'(\vec{e}_i) \to \exists \overrightarrow{e_j : \tau_j}.\, H'(\vec{e}_i, \vec{e}_j)$$

If $G' \to G$ and $H \to H'$, then S_1 subsumes S_2. In this case, Specy keeps S_1 and prunes S_2. We use syntactic checking first because it is very fast and does not require an SMT solver (e.g., Z3 [11]), although it can miss some semantic implications due to over-approximation in the abstraction of predicates.

Pruning by semantic checking. To detect implications missed by syntactic checking, Specy leverages Z3 [11] to check implications $G \to H$, and then uses them for detecting 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.

Pruning symmetric formulas. 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 Specy only keeps one of them:

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

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 specifications and check them on a protocol model by utilizing PChecker [12]. Since pruning by PChecker often requires more time than logic-based pruning, this step is optional.