



SPECY: Learning Specifications for Distributed Systems from Event Traces

MIKE HE, Princeton University, USA

ANKUSH DESAI*, Snowflake, USA

AISHWARYA JAGARAPU, Amazon Web Services, USA

DOUG TERRY*, LinkedIn, USA

SHARAD MALIK, Princeton University, USA

AARTI GUPTA, Princeton University, USA

Reasoning about the correctness of distributed systems is a significant challenge, with precise correctness specifications serving as an essential prerequisite to verification. However, identifying and formulating specifications remains a major hurdle for developers in practice. SPECY addresses this challenge by automatically learning specifications from observable *event traces* generated by message exchanges in distributed systems. The system employs a specialized grammar tailored for event-based specifications, incorporating support for quantifiers over events – a capability essential for capturing the complex behavioral patterns inherent in distributed protocols. SPECY utilizes a novel learning procedure that combines grammar-based enumerative search with dynamic learning from event traces, providing effective control over the specification search. We evaluated SPECY on established distributed protocols and industrial case studies, demonstrating its ability to successfully learn important protocol specifications. SPECY can discover previously unidentified specifications overlooked by developers, automatically derive inductive invariants that were previously constructed manually for verification purposes, and, through run-time monitoring in production systems, reveal gaps in testing coverage – highlighting opportunities to leverage specifications in practice.

CCS Concepts: • **Theory of computation** → **Program specifications**; • **Information systems** → *Parallel and distributed DBMSs*; • **Software and its engineering** → **Software verification and validation**.

Additional Key Words and Phrases: specification inference, distributed systems, automated reasoning

ACM Reference Format:

Mike He, Ankush Desai, Aishwarya Jagarapu, Doug Terry, Sharad Malik, and Aarti Gupta. 2026. SPECY: Learning Specifications for Distributed Systems from Event Traces. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 101 (April 2026), 29 pages. <https://doi.org/10.1145/3798209>

1 Introduction

Developers of distributed systems face the formidable challenge of reasoning about system correctness in the presence of myriad interleavings of messages and potential failures. A fundamental prerequisite to achieving such rigorous reasoning is the formulation and comprehension of the *correctness specifications* themselves. These specifications, which are typically expressed as safety

*Most of the work was done when the authors were at Amazon Web Services.

Authors' Contact Information: Mike He, Princeton University, Princeton, USA, mikehe@princeton.edu; Ankush Desai, Snowflake, Menlo Park, USA, Ankush.desai@snowflake.com; Aishwarya Jagarapu, Amazon Web Services, Seattle, USA, aishuj@amazon.com; Doug Terry, LinkedIn, San Francisco, USA, douglasbterry@hotmail.com; Sharad Malik, Princeton University, Princeton, USA, sharad@princeton.edu; Aarti Gupta, Princeton University, Princeton, USA, aartig@princeton.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/4-ART101

<https://doi.org/10.1145/3798209>

or liveness properties, provide the essential foundation for diverse validation approaches employed across industry and academia – ranging from lightweight testing [16, 48, 64] or state space exploration via model checking [13, 26, 33], to formal proofs using interactive theorem proving [27, 39, 49, 50, 55] or (semi-)automated deductive verification [23, 45, 46, 62, 63]. Furthermore, the importance of formal specifications is also well-recognized in industry practice [6, 9, 35, 43].

Our largely anecdotal experience with the open-source P modeling framework [4, 13] for reasoning about industry-strength distributed services has revealed that formulating correctness specifications that are useful across development stages imposes a significant and time-consuming burden on service teams. Developers are required to articulate specifications that can be applied across diverse tasks, including formal verification, runtime monitoring, and testing. To successfully alleviate this burden, there is a clear demand for a *scalable* and *automated* specification learning framework. Such a framework must target specifications that are (1) capable of encoding the correctness conditions developers may require, and (2) useful across different downstream tasks, from model checking or deductive verification, to finally runtime monitoring in production, which serves as the last resort for guarding service quality.

While a variety of existing techniques address the challenge of learning specifications and invariants for distributed systems [8, 19, 21, 40, 56, 59–61], they only partially fulfill the above demands. These methods can be broadly classified into two main approaches: (1) verifier-aided synthesis [2, 3, 21, 60–63] and (2) data-driven dynamic learning [19, 56, 59]. Approaches from the first category, exemplified by DuoAI [60] and SWISS [21], can synthesize expressive invariants (including $\forall\exists$ quantifier patterns); however, their practical application faces considerable challenges. These tools are explicitly designed to find *inductive invariants* and require models expressed in decidable logic fragments [45], which involve significant intellectual and manual effort upfront. They are not designed to find safety specifications that are not inductive by themselves, and suffer from inherent limitations in verifier performance and scalability. In contrast, data-driven dynamic learning techniques such as Dinv [19] and LIDO [56] avoid the need for upfront development of decidable models, but typically find invariants in fixed-size system configurations, encoding them as formulas that are *quantifier-free*. They do not support complex quantification patterns like $\forall\exists$ alternation, which are necessary to express many specifications [11, 44, 45, 53]. They thus require users to manually add quantifiers for generalization to arbitrary-sized configurations – a process that is both burdensome and error-prone. A further critical limitation common to both verifier-aided [21, 46, 60, 62, 63] and data-driven approaches [19, 56, 59] is that they target learning invariants on *global states*, that do not directly capture temporal and payload relationships between message events. However, tracking of global states in distributed systems is computationally expensive. Therefore, the invariants discovered by these approaches are limited in utility across different stages of development. While they can be used with *models* with global states, they are impractical for use with *production-scale deployments*, since one would have to re-construct or synchronize global states from distributed local states or from exchanged messages.

In this work, we set our goal to *automatically learn quantified specifications (including \forall and $\forall\exists$ quantifier patterns) over message events, that can be used in multiple development stages, and with scalable performance on complex large-scale systems*. We achieve this goal in SPECY by developing some key ideas highlighted below.

Learning event-based specifications. To learn specifications suitable for both verification and monitoring of production-scale systems, we shift the focus away from state-based specifications (targeted in many prior works) to *event-based* ones that express relationships between events generated by message exchanges in distributed systems. We focus especially on *causally-related events*, since their payloads capture relationships that are important for expressing correctness

Table 1. Example event-based specifications, with $\boxed{\text{Guards } G}$, $\boxed{\text{Witnesses } W}$, $\boxed{\text{Hypotheses } H}$. User guidance is needed in some examples, e.g., \preceq_u of *Update Prop.* is a *user-defined* predicate over log entries.

Specification Description	Specification Formula
<i>External Consistency</i> (Spanner [11])	$\forall e_0, e_1 : e\text{Commit. } \boxed{e_0.\text{commit_time} < e_1.\text{start_time}} \rightarrow e_0.\text{ts} < e_1.\text{ts}$
<i>Election Safety</i> (Raft [44])	$\forall e_0, e_1 : e\text{Elected. } \boxed{e_0.\text{term} = e_1.\text{term}} \rightarrow e_0.\text{leader} = e_1.\text{leader}$
<i>Update Prop.</i> (Chain Replication [53])	$\forall e_0, e_1 : e\text{NodeLog. } \boxed{e_0.\text{pos} \leq e_1.\text{pos}} \rightarrow e_1.\text{log} \preceq_u e_0.\text{log}$
<i>Whitelist Safety</i> (Firewall [42])	$\forall e_0 : e\text{Recv. } \boxed{e_0.\text{allowed} = \top} \rightarrow \exists e_1 : e\text{Grant. } \boxed{e_1 \prec e_0} \wedge e_0.\text{src} = e_1.\text{host}$
<i>Quorum Votes</i> (Consensus [41])	$\forall e_0 : e\text{Decide. } \exists_{\geq \text{quorum}} e_1 : e\text{Vote. } \boxed{e_1 \prec e_0} \wedge e_0.\text{ballot} = e_1.\text{vote}$

properties, and this avoids considering irrelevant message orderings in the system. Event-based specifications have important benefits. They can be monitored at the message interfaces of individual system components, and do not require a centralized view or tracking of the global system state. Plus, they can be applied across different levels of abstraction to support refinement and conformance checking, from high-level models down to code-level implementations, so long as the component message interfaces remain the same. Users may also extend the message interface (via code and/or instrumentation) to selectively include shared states to guide the learning of state-based specifications. Also, recent advances in protocol verifiers [42] target event-based specifications, and SPECY can be used to find inductive invariants to assist the proofs with such verifiers.

Grammar for correctness specifications (§3). SPECY learns specifications in first-order logic (FOL) with formulas that satisfy a custom grammar tailored for distributed systems. Our custom grammar is *more general and flexible* than prior works in dynamic learning [8, 19, 40, 56, 57], which focused on temporal relations or pre-defined properties, with little support for quantifiers. Our grammar allows logical constraints to be expressed over all events of a certain type, or *relational* constraints over multiple events of the same/different types, where each event is associated with a message and its payload. The former are useful for capturing local specifications (for a single component), and the latter are useful for capturing inter-component relational specifications. It allows \forall and $\forall\exists$ quantifier patterns on events and quorum-based quantification.

The custom grammar in SPECY is based on our insight that correctness specifications for most practical protocols follow an *if-guard-then-witness-and-relation* pattern; some examples from well-known distributed system protocols are shown in Table 1. Each specification has: (1) typed events, which are quantified, (2) *Guards* G , predicates on events or their payloads to capture *control conditions*, (3) *Witness* W (optional), predicates that capture *existence conditions* for witness events, and (4) *Hypotheses* H , predicates that express relational conditions (often on payloads) that *must* hold under the Guard and Witness conditions. For instance, the *Whitelist Safety* specification of a Firewall protocol [42] states that if $\boxed{\text{receiving is allowed in event } e_0}$ (guard G), then there must *exist* an event e_1 that $\boxed{\text{happened before}}$ (witness W), and which $\boxed{\text{grants access to the host which is the sender in } e_0}$ (hypothesis H). More generally, the Witnesses enable SPECY to learn specifications with *quorum constraints* on the number of witnesses (e.g., *Quorum Votes* in Table 1). In addition to a common set of predicates provided by SPECY, users may also extend this set by defining other customized predicates in code (e.g., \preceq_u of *Update Prop.* in Table 1).

Fine-grained control in learning from event traces (§4). SPECY uses the custom grammar to constrain the search space of candidate specifications during learning, and to *decompose* the search problem into a novel combination of static enumerative search and dynamic learning over event traces. Essentially, it performs a grid point search where each point statically selects: (i) the quantified events and pattern (called an event combination), and (ii) all combinations (up to a

certain number) of allowed predicates that can appear in G and W . Then SPECY invokes a dynamic learner – Daikon [15] in our prototype – to learn the predicates in H . Notably, *the grammar and the static enumerative search provide a fine-grained control over the scope of dynamic learning*. Plus, SPECY performs several pruning steps after dynamic learning to reduce the number of learned specifications. Both of these strategies help to avoid learning irrelevant and trivial specifications, which is in general a big challenge in dynamic trace-driven approaches.

Impact of learned specifications in downstream tasks (§5). The specifications learned by SPECY can be used in downstream tasks to provide benefits in development practice in (at least) three ways. First, learned specifications can be used to formally prove correctness using a verifier. We show (§6.4) that SPECY can learn both safety properties and inductive invariants needed by an automated verifier [42] to provide proofs for many benchmarks. Second, SPECY automatically translates the learned specifications to executable runtime monitors, which can be applied to monitor production traces, to check whether the real-world implementation adheres to the formal model (shown in §6.5.1). Third, presenting the learned specifications to developers can reveal gaps in specifications and behavioral test coverage. In our evaluations (§6.5.2), we present a case study where SPECY discovers important specifications overlooked by developers. In addition, when the learned specifications are used as runtime monitors on production traces, we identified an important gap in the behavioral test coverage of the production system.

Summary of Contributions

(1) We present SPECY, a novel framework that: (a) automatically learns correctness specifications (including quantifiers, $\forall\exists$ alternation, and quorum constraints) (b) from observable execution traces of messages, (c) without relying on decidable models and verifiers, i.e., where the model does not need to be written in a decidable logic upfront, and where learning the specifications does not require using a verifier.

(2) We introduce a novel and effective learning procedure targeting quantified event-based specifications. It combines static enumerative search and dynamic learning on event traces, with fine-grained control on the scope of dynamic learning to avoid learning irrelevant and trivial specifications.

(3) We show the benefits of SPECY in practice: learning complex specifications for a suite of well-known and industrial scale protocols, learning inductive invariants for a downstream verifier, and automatically generating runtime monitors for production deployments that reveal gaps in testing coverage and developers’ understanding of system behaviors.

2 Overview of SPECY Framework

In this section, we first introduce the P language using a running example. Then we provide an overview of the custom grammar that SPECY uses to learn specifications. Finally, we present the workflow used by SPECY for learning specifications from event traces.

2.1 Background: P Modeling Language

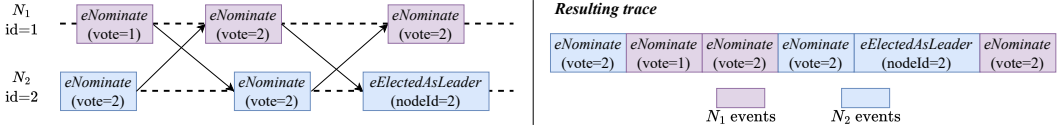
P [4, 13] is a state-machine-based programming language designed for formal modeling and analysis of distributed systems. It enables developers to model system designs as collections of communicating state machines, with support for checking safety and liveness properties. For instance, Amazon Web Services (AWS) used P to validate the strong consistency protocol in Amazon S3, gaining confidence in its correctness during the transition from eventual to strong consistency [9]. Similarly, DeepSeek employed P to check the correctness of the Fire-Flyer File System (3FS), a high-performance distributed file system [5], showcasing its effectiveness in analyzing complex distributed systems.

```

1 type tNodeId = int;
2 event eNominate: (vote: tNodeId);
3 event eElectedAsLeader: (nodeId: tNodeId);
4
5 machine Node {
6   # unique node identifier
7   var id: tNodeId;
8   # node to its right
9   var right: Node;
10
11  start state Init {
12    entry (cfg: (id: tNodeId, nxt: Node)) {
13      id = cfg.id;
14      right = cfg.nxt;
15      goto Nominating;
16    }
17  }
18
19  state Nominating {
20    entry {
21      send right, eNominate, (vote=id,);
22    }
23    on eNominate do (p: (vote: tNodeId)) {
24      if (p.vote == id) {
25        announce eElectedAsLeader, (nodeId=id,);
26        goto Won;
27      } else if (p.vote > id) {
28        send right, eNominate, (vote=p.vote,);
29      } else {
30        send right, eNominate, (vote=id,);
31      }
32    }
33    state Won { ignore eNominate; }
34  }

```

Fig. 1. Ring Leader Election (RLE) protocol modeled in P

Fig. 2. An example event trace for RLE protocol with 2 nodes N_1 and N_2

2.2 Running Example: Ring Leader Election Protocol

A P program comprises state machines communicating asynchronously with each other using events for exchanging messages with typed payloads. 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 (RLE) [10] as our running example. In this protocol, nodes are arranged in a ring topology and assigned a unique, totally ordered identifier (id). To initiate the leader election, each node sends its id to the node on its right. Upon receiving an id, a node compares it with its own id. If they are equal, the node declares itself the leader. Otherwise, it forwards the greater id to the node on its right. Ultimately, the node with the highest id elects itself as the *unique* leader.

Fig. 1 shows the state machine Node (line 5) in the RLE protocol. The event declarations (lines 2, 3) specify the names and payload types of the events used in the RLE 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 22), 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 20) or **announce** (line 24), respectively. Executing **goto** statements (e.g., line 15) 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

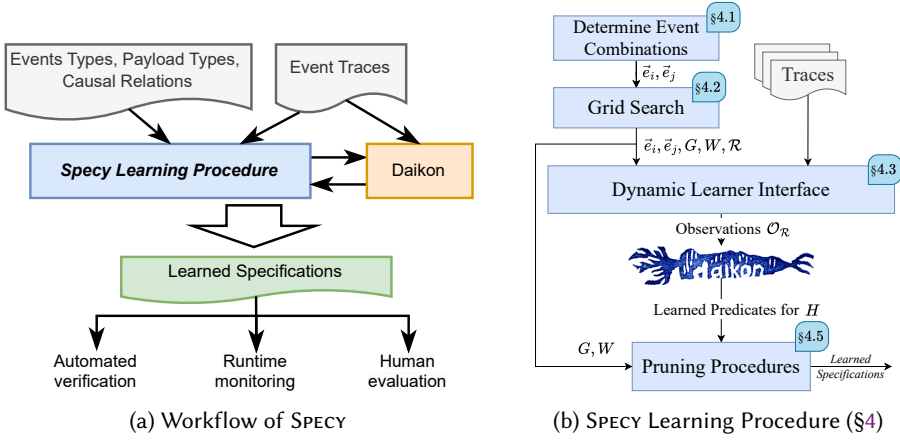


Fig. 3. Top-level workflow of SPECY (left) and the Learning procedure (right). The learning procedure determines event combinations, combines a static enumerative grid search with Daikon-based dynamic learning, and performs pruning.

eNominate event to right with the greater id as the payload. Fig. 2 shows an example execution with two nodes (left), and the associated event trace (right), over which SPECY performs learning.

2.3 Custom Grammar for Event-Based Specifications

To learn specifications that are useful for distributed systems such as the Ring Leader Election protocol modeled in our running example, SPECY uses the following *formula template* for specification ϕ , which can capture many correctness properties (as shown in Table 1, and later in §6):

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

The formula uses universal quantifiers (\forall , at least one) as well as nested existential quantifiers (\exists , optionally) over vectors of typed events (\vec{e}_i, \vec{e}_j , respectively). Formulas with single event variables are useful for expressing local component-based specifications, while formulas with two (or more) event variables capture inter-component relational specifications. Although the grammar supports quantification over any fixed number of event types, we found that it is enough in practice to learn specifications that are quantified over just *two* types of events that are causally-related.

Also, note that this grammar does not support a $\exists\forall$ quantifier pattern. For example, in network routing protocols, one may wish to express a unique route convergence property: there *exists* a unique routing solution at protocol convergence, such that *all* routing messages received after convergence result in the same routing solution. However, we found that most correctness specifications do not fall in this pattern. We will discuss how SPECY can be extended to support other patterns in §6.6.

At a high level, the quantifier-free part of the formula template uses the if-then form (expressed using implication \rightarrow) that occurs commonly in specifications to capture conditions under which certain relationships hold on events. Accordingly, it differentiates the subformulas shown as Guard (\boxed{G}), Witness (\boxed{W}), and Hypothesis (H). Note that when G is true (\top), the W (when specified) and H must hold unconditionally. For formulas with single event variables, setting G to true can express properties that avoid bad (i.e., unsafe) messages.

The grammar allows temporal (e.g., happens-before [32]) and payload relationships (e.g., equality, arithmetic) in the G, W, H subformulas, via standard (as well as user-defined) functions and

predicates over events and their typed fields. Although this formula template resembles a popular decidable fragment of FOL called EPR [46], SPECY does not rely on availability of decidable models or verifiers for learning specifications. The main requirement is that all function and predicate symbols used in the specification grammar must have standard (or user-defined) evaluators that can be applied on concrete events in the traces.

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 largely automated, with the ability to leverage user guidance. 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.

2.4 From Event Traces to Specifications

We now elaborate on how SPECY is used for learning specifications and how the learned specifications are applied in practice. The high-level workflow is shown in Fig. 3a, with the learning procedure components shown in Fig. 3b (detailed later, §4).

① **Prepare inputs.** The user first prepares inputs to SPECY: (1) a set of event traces (e.g., by using a P model and the PChecker [4] to generate traces), and (2) trace metadata including event types, payload types, and causal relations between events. Note that while a P model is useful for automatically extracting these inputs, it is not required for SPECY to learn specifications. When the P model is not available, users need to provide these inputs as configurations to SPECY.

② **Computing grammars (§3) and learning specifications (§4).** SPECY uses the metadata to identify which events and quantifiers to target in the specification formula template $\phi(\vec{e}_i, \vec{e}_j$ in Eqn 1) based on causality patterns in sending/receiving messages. For each such combination, SPECY computes a grammar that generates the set of allowed *terms* and *predicates* in ϕ . To learn ϕ from traces, SPECY *statically enumerates* G (and optionally W) from the grammar, and uses Daikon [15] to *dynamically learn* H over terms from events in traces where G (and W) are true. Finally, SPECY *assembles* the learned specification ϕ by putting together these parts.

③ **Effective pruning (§4.5).** Next, SPECY applies several logic-based pruning steps to remove *redundant* specifications (e.g., tautologies, formulas that are subsumed by other formulas). If a P model is available, SPECY translates the learned specifications into PChecker assertions and applies PChecker to remove specifications that are falsified by the model checker.

Running example. For the RLE protocol, SPECY successfully learns the well-known safety specification as formulas over the *eElectedAsLeader* (\mathcal{E}) and *eNominate* (\mathcal{N}) events shown below. These formulas together state that a *unique* leader (Eqn 1) with the *highest* id (Eqn 2) is *elected* (Eqn 3). Here, $e_1 \prec e_0$ encodes the standard *happens-before* relation [32]. When interpreted over a single trace, an event e_1 happens-before another event e_0 if e_1 appears earlier in the trace than e_0 .

$$\forall e_0, e_1 : \mathcal{E}. e_0.\text{nodeId} = e_1.\text{nodeId} \quad (1)$$

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

$$\forall e_0 : \mathcal{E}. \exists e_1 : \mathcal{N}. e_1 \prec e_0 \wedge e_0.\text{nodeId} = e_1.\text{vote} \quad (3)$$

Application of learned specifications. For our running example, SPECY learns 30 specifications in total, including the above three safety properties. In addition, SPECY learns three inductive invariants (§6.4) that were previously provided manually in the proof of the RLE protocol. These

Vars	$v \in V$
Constants	$c \in C$
Event Vars	$e \in E$
Event Types	τ_i
Terms	$T ::= T_E \mid T_F$
Event Terms	$T_E ::= \text{indexof}(E)$
Field Terms	$T_F ::= C \mid E.v \mid f(T_F^+)$
Preds	$P ::= P_E \mid P_F \mid \neg P_F$
Event Preds	$P_E ::= T_E = T_E \mid T_E < T_E \mid T_E > T_E$
Field Preds	$P_F ::= T_F = T_F \mid T_F < T_F$ $\mid T_F \leq T_F \mid uP(uf(T_F^+)^+)$
SetCardinality	$SC(m) ::= m \geq 1 \mid m = C \mid m \leq C \mid m \geq C$

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

Fig. 4. Grammar for Specification Formulas

help the verifier [42] complete the proof of the safety properties *automatically*. Other downstream applications like runtime monitoring are demonstrated in our evaluations (§6.5.2).

3 Custom Grammar in SPECY

In this section, we formally define the grammar and semantics for the specification formula template used in SPECY. We provide examples that demonstrate its expressiveness and also highlight user guidance that can customize the search for candidate specifications.

3.1 Grammar and Semantics

SPECY targets two top-level rules in formula templates, shown as boxed rules in the grammar listed in Fig. 4. The first boxed rule, shown as ϕ_{\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 $(\boxed{G}, \boxed{W}, \boxed{H})$ is a conjunction of *atomic predicates* P , defined over events or terms. The grammar for terms (Terms T) and predicates (Preds P) is shown above the box in Fig. 4. 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 either Event Terms (T_E) or Field Terms (T_F). Event Terms are special terms with application of `indexof` function on an event variable, which is interpreted as the index of that event in a trace. Field Terms are comprised of constants (C), fields of event payloads ($E.v$) and function applications over Field Terms ($f(T_F^+)$; here $+$ means one or more). For function applications over Field Terms ($f(T_F^+)$), SPECY

supports standard arithmetic functions over integers and floating points, and a `sizeof` function for a list or set that returns its size.

Predicates (P) are also of two kinds: (1) Event Preds (P_E) that relate Event Terms, and (2) Field Preds ($P_F, \neg P_F$) that relate Field Terms. The former includes arithmetic comparison over `indexof(E)` that encodes the standard happens-before (\prec) relation [32] 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 $uf(T^+)^+$). 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 Universal Quantification. Our grammar can express formulas that universally quantify over one or more types of events (ϕ_V). This pattern captures local properties when quantifying over a single type of event (e.g., for all decision events, the round numbers are valid). When quantifying over multiple types of events, it captures relational specifications that express cross-event properties (e.g., all read events carry the highest version number among all previous writes to the same key).

Support for Existential Quantification. We also support a forall-exists ($\forall\exists$) quantifier pattern in $\phi_{V\exists}$, 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 $SC(m)$ constraint as an *augmented existential quantifier* — it allows quantification over a *set* of witness events, where the cardinality m of the witness set satisfies the labeled SC constraint. This is very useful for expressing quorums in consensus protocols. The standard semantics of an existential quantifier is equivalent to the SC constraint: $m \geq 1$.

Expressiveness. We use the grammar defined in Fig. 4 to restrict the candidate space during learning, while still allowing useful formulas. We show in our evaluations that it captures safety specifications of many well-known as well as proprietary protocols (§6). We made some tradeoffs in expressiveness to gain search efficiency. One tradeoff is that SPECY does not currently support specifications with a $\exists\forall$ quantifier pattern, e.g., the unique convergence property of network routing protocols (§2.3).

Implicit Disjunctions. Another tradeoff is in allowing only implicit, rather than explicit, disjunctions. Specifically, SPECY allows a top-level implication (which is implicitly a disjunction), where the constituents G, W, H are conjunctions of atomic predicates. Note that disjunctions can be implicitly expressed in these constituents: for disjunctions in $G = G_1 \vee G_2$, SPECY considers G_1, G_2 separately. If the same H is learned under both, it corresponds to learning H under the disjunct G . For W , the user can define a new predicate as the disjunction of other atomic predicates. For a learned H , SPECY does not directly control the predicates in H , and the dynamic learner may support predicates that implicitly encode disjunctions. For example, the set-membership predicate `member(x, S)` supported by Daikon encodes the disjunction $\bigvee_{s \in S} x = s$.

Semantics of Specification Formulas over Traces. The specification formulas are interpreted over the given set of event traces, and the semantics are fairly straight-forward where the quantified events are interpreted over all type-consistent instantiations in the traces. Selected semantic rules for interpretation over a single trace \mathcal{T} are shown in Fig. 5 (with the full set in Appendix B). We extend the semantics to a set of traces \mathbb{T} , if the specification formula is true for each $\mathcal{T} \in \mathbb{T}$.

Formally, we interpret terms (T), predicates (P), and formulas (ϕ) over a trace \mathcal{T} , an *event typing context* Γ , and an *event instantiation* σ . A trace \mathcal{T} is a finite sequence of events ordered by their occurrence. A typing context Γ maps event variables E to types τ , and we use $[e \mapsto \tau]\Gamma$ to denote

TRACES \mathcal{T} , EVENT TYPE CONTEXT Γ , EVENT INSTANTIATIONS σ , SET OF INSTANTIATIONS Σ

$$\mathcal{T} \triangleq \langle z_0 : \tau_0, z_1 : \tau_1, \dots, z_n : \tau_n \rangle \quad \Gamma : E \rightarrow \tau \quad \sigma : E \rightarrow z \in \mathcal{T} \quad \Sigma_{e:\tau} \triangleq \{ \{e \mapsto z_i\} \mid z_i : \tau \in \mathcal{T} \}$$

SEMANTICS OF TERMS (T) OVER \mathcal{T} , Γ , σ

$$\begin{array}{c} \text{EVAL-INDEX} \\ \mathcal{T} \quad \Gamma(e) = \tau_i \quad \sigma(e) = z_i \quad z_i : \tau_i \in \mathcal{T} \\ \hline \text{EVAL}(\sigma, \text{indexof}(e)) = i \end{array} \qquad \begin{array}{c} \text{EVAL-PROJ} \\ \mathcal{T} \quad \Gamma(e) = \tau_i \quad \sigma(e) = z_i \quad z_i : \tau_o \in \mathcal{T} \\ \hline \text{EVAL}(\sigma, e.v) = z_i.v \end{array}$$

$$\begin{array}{c} \text{EVAL-FUN} \\ \mathcal{T}, \Gamma \quad \text{EVAL}(\sigma, t_0) = v_0 \quad \dots \quad \text{EVAL}(\sigma, t_n) = v_n \\ \hline \text{EVAL}(\sigma, f(t_0, \dots, t_n)) = f(v_0, \dots, v_n) \end{array}$$

SEMANTICS OF ATOMIC PREDICATES (P) AND BOOLEAN COMBINATIONS OVER \mathcal{T} , Γ , σ

$$\begin{array}{c} \text{SEM-T} \\ \mathcal{T}, \Gamma, \sigma \quad \text{EVAL}(\sigma, p) = \top \\ \hline \mathcal{T}, \Gamma, \sigma \models p \end{array} \qquad \begin{array}{c} \text{SEM-}\perp \\ \mathcal{T}, \Gamma, \sigma \quad \text{EVAL}(\sigma, p) = \perp \\ \hline \mathcal{T}, \Gamma, \sigma \models \neg p \end{array}$$

$$\begin{array}{c} \text{SEM-AND} \\ \mathcal{T}, \Gamma, \sigma \models Q_1 \quad \mathcal{T}, \Gamma, \sigma \models Q_2 \\ \hline \mathcal{T}, \sigma \models Q_1 \wedge Q_2 \end{array} \qquad \begin{array}{c} \text{SEM-IMP-L} \\ \mathcal{T}, \Gamma, \sigma \models \neg Q_1 \\ \hline \mathcal{T}, \Gamma, \sigma \models Q_1 \rightarrow Q_2 \end{array}$$

$$\begin{array}{c} \text{SEM-IMP-R} \\ \mathcal{T}, \Gamma, \sigma \models Q_2 \\ \hline \mathcal{T}, \Gamma, \sigma \models Q_1 \rightarrow Q_2 \end{array} \qquad \begin{array}{c} \text{SEM-TEMPORAL-HB} \\ \mathcal{T}, \Gamma, \sigma \quad \text{EVAL}(\sigma, \text{indexof}(e)) < \text{EVAL}(\sigma, \text{indexof}(e')) \\ \hline \mathcal{T}, \Gamma, \sigma \models \text{indexof}(e) < \text{indexof}(e') \end{array}$$

SEMANTICS OF SPECIFICATION FORMULAS (ϕ) OVER \mathcal{T} , Γ , σ

$$\begin{array}{c} \text{SEM-}\forall \\ \text{for each } \sigma_i \in \Sigma_{e:\tau}: \mathcal{T}, [e \mapsto \tau]\Gamma, \sigma_i \cup \sigma \models \phi \\ \hline \mathcal{T}, \Gamma, \sigma \models \forall e : \tau. \phi \end{array} \qquad \begin{array}{c} \text{SEM-}\exists \\ \text{for some } \sigma_i \in \Sigma_{e:\tau}: \mathcal{T}, [e \mapsto \tau]\Gamma, \sigma_i \cup \sigma \models \phi \\ \hline \mathcal{T}, \Gamma, \sigma \models \exists e : \tau. \phi \end{array}$$

$$\begin{array}{c} \text{SEM-}\exists_{SC} \\ SC(m) \text{ is true, for } m \text{ distinct } \sigma_i \in \Sigma_{e:\tau}: \mathcal{T}, [e \mapsto \tau]\Gamma, \sigma_i \cup \sigma \models \phi \\ \hline \mathcal{T}, \Gamma, \sigma \models \exists_{SC} e : \tau. \phi \end{array}$$

Fig. 5. Semantics of specification formulas (selected rules)

extending Γ with a type mapping $e \mapsto \tau$. An event instantiation σ maps event variables $e_i \in E$ to type-consistent concrete events z_i in the trace. Essentially, σ is an interpretation that maps event variables in the formulas to concrete events in the trace (i.e., the domain of interpretation).

Given \mathcal{T} , Γ , and σ , the semantics of terms is their *evaluation* based on σ , denoted $\text{EVAL}(\sigma, T)$. For instance, rule EVAL-INDEX states that $\text{indexof}(e)$ evaluates to i if σ maps e to the i^{th} event z_i in \mathcal{T} , and Γ maps e to the type of z_i . We also allow user-defined functions in terms, where function applications evaluate to some concrete values (via EVAL-FUN). The semantics of *atomic* predicates (e.g., $T_F = T_F$) is defined by the truth values of their evaluations based on involved terms. For instance, the *happens-before* relation [32] semantics is shown in the rule SEM-TEMPORAL-HB, based on evaluations of $\text{indexof}(e)$ and $\text{indexof}(e')$. Similarly, we allow user-defined predicates that can be evaluated to \top or \perp under this semantics. (Other rules are shown in Appendix B.) Semantics of logical connectives is inductively defined (e.g., SEM-AND) based on sub-formulas. Semantics of quantified specification formulas are defined by rules SEM- \forall , SEM- \exists and SEM- \exists_{SC} . A formula $\forall e : \tau. \phi$ is true for \mathcal{T} , Γ , σ , if *for every* instantiation σ_i with $\sigma_i(e) = z_i : \tau \in \mathcal{T}$, ϕ holds on \mathcal{T} ,

$[e \mapsto \tau]\Gamma, \sigma_i \cup \sigma$. Similarly, $\exists e : \tau. \phi$ holds if ϕ holds for *at least one* instantiation; a formula with \exists_{SC} holds if the SC constraint holds for m and there are m distinct instantiations where ϕ holds.

3.2 Examples of Specification Formulas

We now explain the example specifications in Table 1 and show how our grammar captures them. We mark the key components, guards \boxed{G} , witnesses \boxed{W} , and hypotheses \boxed{H} , involved in each specification. The upper set of specifications in Table 1 shows examples that use only forall (\forall) quantifiers, captured by ϕ_{\forall} in the grammar. Predicates in these specifications are captured by Field Preds (P_F) in our grammar.

<p><i>External Consistency</i> of Google Spanner [11].</p> <p>\boxed{G} if a transaction commits before another transaction starts, then</p> <p>\boxed{H} its commit timestamp is less than the commit timestamp of the other transaction.</p> <p><i>Election Safety</i> of the Raft [44] protocol. It states the <i>uniqueness of elected leader</i> in a single term.</p> <p>\boxed{G} if the term number is the same in two become-leader events, then</p> <p>\boxed{H} the event is sent by the same elected leader.</p> <p><i>Update Propagation</i> specification of Chain Replication [53].</p> <p>\boxed{G} if a node n_0 is placed before n_1 in the chain, then</p> <p>\boxed{H} the commit log on n_1 is a prefix (\preceq_u) of the commit log on n_0.</p>

Note that the Update Propagation specification has a user-defined predicate \preceq_u .

The bottom set shows two specifications that require $\forall\exists$ alternation of quantifiers in $\phi_{\forall\exists}$.

<p><i>Whitelist Safety</i> of a Firewall protocol [42] (explained in §1).</p> <p>\boxed{G} if receiving is allowed then</p> <p>\boxed{W} there exists an event that <i>happened before</i></p> <p>\boxed{H} that grants access to the host.</p> <p><i>Quorum Votes</i> of a Consensus protocol [41].</p> <p>\boxed{G} for all decisions (τ)</p> <p>\boxed{W} there exists at least <i>quorum</i> distinct votes that <i>happened before</i></p> <p>\boxed{H} that voted for the decision.</p>

Note that the *quorum* constraint is supported by an augmented existential quantifier with the SC constraint, where the number of votes for a decision must be \geq *quorum*.

3.3 User Guidance for Customizing Grammar

The grammar shown in Fig. 4 can be customized by a user, by providing configuration inputs or by adding code. We categorize different levels of user guidance (labeled UG1, UG2, UG3 below) that enhance the search space and identify domain/protocol-specific relationships between events.

(UG1) Event combinations. A user can identify interesting event combinations over which SPECY should learn the specifications. For instance, a user can specify an event combination $\vec{e}_i = \langle \mathcal{N} \rangle$ and $\vec{e}_j = \langle \mathcal{E} \rangle$ as a configuration input in SPECY, to enable SPECY to learn a $\phi_{\forall\exists}$ formula, $\forall e_i : \mathcal{N}. G(e_i) \rightarrow \exists e_j : \mathcal{E}. W(e_i, e_j) \wedge H(e_i, e_j)$.

Table 2. Causal relation patterns that result in the listed Event Combinations ($\forall e_i : \tau_i$ and $\exists e_j : \tau_j$). For each identified causal relation, a \forall pattern is always selected; an *empty* indicates that the $\forall\exists$ pattern is not selected. Note that the Send-then-listen causal relation does not appear (indicated N/A) in the RLE example.

Causal Relation Patterns	Event Comb.		RLE example (Fig.1)	Lines	$\forall \vec{e}_i : \vec{\tau}_i$	$\exists \vec{e}_j : \vec{\tau}_j$
	$\forall \vec{e}_i : \vec{\tau}_i$	$\exists \vec{e}_j : \vec{\tau}_j$				
Same-sourced-sends (sends e_0 , sends e_1)	$\langle \tau_0, \tau_1 \rangle$	empty	Same-sourced-sends	24	$\langle \mathcal{E}, \mathcal{E} \rangle$	empty
Receive-then-send (receives e_0 , then sends e_1)	$\langle \tau_0, \tau_1 \rangle$	empty		24, 27, 29	$\langle \mathcal{E}, \mathcal{N} \rangle$	empty
	$\langle \tau_1 \rangle$	$\langle \tau_0 \rangle$		20, 27, 29	$\langle \mathcal{N}, \mathcal{N} \rangle$	empty
Send-then-listen (sends e_0 , then listens to e_1)	$\langle \tau_0, \tau_1 \rangle$	empty	Receive-then-send	22, 24	$\langle \mathcal{N}, \mathcal{E} \rangle$	empty
	$\langle \tau_0 \rangle$	$\langle \tau_1 \rangle$			$\langle \mathcal{E} \rangle$	$\langle \mathcal{N} \rangle$
			Send-then-listen	N/A	-	-

(UG2) Custom predicates. A user can add *ghost code* to define complex predicates that can appear in the target formula. For example, a prefix relationship \preceq_u between lists (shown in *Update Prop.* in Table 1) can be provided as a custom predicate over two lists l_1, l_2 , defined as $l_1 \preceq_u l_2 \iff |l_1| \leq |l_2| \wedge \forall i \in [0, |l_1|). l_1[i] = l_2[i]$. Again, the main requirement is that custom functions and predicates must be supported by evaluators.

(UG3) Exposing system states. A user can add ghost code or instrument the system to expose some system state as auxiliary payload fields in events. For instance, in our running example (§2.2), a user may choose to include the local variable `right` into the payload field of the *eNominate* event. We show that this type of user guidance is helpful for SPECY to learn inductive invariants (§6.4).

4 Learning Procedure in SPECY

In this section, we describe the learning procedure in SPECY that starts from the input traces and metadata, and generates the specifications reported to a user, which can then be used in downstream tasks. The important steps and components are shown in Fig. 3b.

4.1 Determining Event Combinations

The first step is to determine the quantified events \vec{e}_i, \vec{e}_j , which we call *event combinations*, in the formula template. 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.

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 driven by common causal relation patterns in protocol specifications. These are summarized in the left part of Table 2, where the first column lists the causal relation patterns, each of which 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 simplicity, we list the event types for each causal pattern). The right part illustrates these patterns on our running example (§2) with the listed event combinations on event types, that result from causal patterns identified on the respective lines in the P program (Fig. 1).

Same-sourced-sends. In this pattern, events with types τ_0 and τ_1 (with possibly $\tau_0 = \tau_1$) are sent in the same entry function or event handler. SPECY will aim 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.

For each selected event combination, SPECY automatically generates a candidate template $CT(\vec{e}_i, \vec{e}_j) = (\mathcal{M}, \mathcal{P})$. Here, \mathcal{M} is a set of terms generated by recursively expanding the “Terms” rule in the grammar (Fig. 4), 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, $\mathcal{M} = \{\text{indexof}(e_0), \text{indexof}(e_1), e_0.\text{nodeId}, e_1.\text{vote}\}$. Similarly, \mathcal{P} is a set of predicates generated by recursively expanding the “Preds” rule in the grammar (Fig. 4), but with restriction to the predicates over the quantified events in \vec{e}_i and \vec{e}_j and their payload fields.

4.2 Enumerative Grid Search

To manage the search space complexity of target formulas, an inner loop performs a *grid search* for each event combination, to learn specifications of increasing complexity that vary in the number of predicates and terms that may appear in the target formula. Specifically, 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. For each grid point and a given Candidate Template $CT(\vec{e}_i, \vec{e}_j) = (\mathcal{M}, \mathcal{P})$, SPECY enumerates different choices for 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 call each such subset of terms as a *Relate Set*, denoted as \mathcal{R} .

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 user-settable bounds in each parameter (where the w parameter is skipped when learning ϕ_v formulas). Note that a grid point with $g = 0$ corresponds to having a guard that is always true, i.e., the consequence of the implication must hold unconditionally.

4.3 Dynamic Learner Interface

For each statically enumerated choice (of G, W , and R), SPECY uses a Dynamic Learner Interface to automatically prepare inputs for a dynamic learner to discover predicates in H . The core semantics of this learning procedure are shown in Fig. 6 (with the complete rules shown in Appendix B).

Filtered traces $\mathcal{T}_{\downarrow\Gamma}$. First, for each event combination (\vec{e}_i, \vec{e}_j) , SPECY *filters* a given trace $(\mathcal{T} \in \mathbb{T})$ by removing events of types that do not match \vec{e}_i or \vec{e}_j , while preserving relevant events and their temporal order, resulting in filtered traces $\mathcal{T}_{\downarrow\Gamma}$, where $\Gamma = \vec{e}_i \cup \vec{e}_j$ is the event typing context that maps quantified events to types (as before in Fig. 5).

Quantifier Instantiation σ . For ease of exposition, we refer to quantified events in 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 $\mathcal{T}_{\downarrow\Gamma}$ and an event type context Γ , SPECY computes *instantiations*, denoted σ (as before), that map symbolic events \vec{e}_i, \vec{e}_j to *all* type-compatible concrete events \vec{z}_i, \vec{z}_j , respectively, that occur within the scope of $\mathcal{T}_{\downarrow\Gamma}$. We use $\mathcal{T}_{\downarrow\Gamma}, \Gamma \triangleright \sigma$ to denote that the instantiation σ results from $\mathcal{T}_{\downarrow\Gamma}$ and Γ . Rule 1 shows how an event in a trace extends the event typing context Γ and the instantiation σ , starting from an initially empty map for each.

Evaluations on trace events and observation δ . Recall that given an event type context Γ and an instantiation σ , we can compute an evaluation of a term T or a predicate P , denoted as $\text{EVAL}(\sigma, T)$, $\text{EVAL}(\sigma, P)$ (Fig. 5). In particular, an evaluation under σ results in a concrete value (e.g., an integer)

TYPES OF FILTERED TRACE $\mathcal{T}_{\downarrow\Gamma}$, OBSERVATIONS OF TERMS δ , WITNESSES OF δ

$$\mathcal{T}_{\downarrow\Gamma} := \langle z_i : \tau_i \mid z_i \in \mathcal{T} \text{ and } \Gamma(e) = \tau_i \text{ for some } e \rangle \quad \delta : T \rightarrow \text{EVAL}(\sigma, T) \quad \omega : \delta \rightarrow 2^\delta$$

(1) INSTANTIATION OF SYMBOLIC EVENTS $e \in \Gamma$ TO CONCRETE EVENTS $z \in \mathcal{T}$

$$\text{INST-INIT} \frac{}{\mathcal{T}, \Gamma_\emptyset \triangleright \sigma_{\text{empty}}} \quad \text{INST-EXTEND} \frac{\mathcal{T}, \Gamma \triangleright \sigma \quad z : \tau \in \mathcal{T}}{\mathcal{T}, [e : \tau]\Gamma \triangleright [e : \tau \mapsto z]\sigma}$$

(2) OBSERVATION δ WHERE G HOLDS

$$\text{CHECK-G} \frac{\mathcal{T}, \Gamma \triangleright \sigma \quad G, \mathcal{R} \quad \mathcal{T}, \Gamma, \sigma \models G}{\mathcal{T}, \Gamma, \mathcal{R}, \sigma \rightsquigarrow_G \underbrace{\{t \mapsto \text{EVAL}(\sigma, t) \mid t \in \mathcal{R}\}}_\delta}$$

(3) WITNESSES $\omega(\delta)$ OF AN OBSERVATION δ WHERE G, W HOLD

$$\text{CHECK-GW} \frac{G, W, \mathcal{R}_{\forall\exists} \quad \Sigma' := \{\sigma' \mid \mathcal{T}_{\downarrow\Gamma_\exists}, \Gamma_\exists \triangleright \sigma' \text{ and } \mathcal{T}, \Gamma_\forall \cup \Gamma_\exists, \sigma \cup \sigma' \models W\} \quad |\Sigma'| > 0}{\mathcal{T}, \Gamma_\forall \cup \Gamma_\exists, \mathcal{R}_{\forall\exists}, \delta \rightsquigarrow_{G,W} \underbrace{\{\{t \mapsto \text{EVAL}(\sigma \cup \sigma', t) \mid t \in \mathcal{R}_{\forall\exists}\} \mid \sigma' \in \Sigma'\}}_{\omega(\delta)}}$$

(4) LEARNING H , GIVEN G, \mathcal{R} (\forall -ONLY)

$$\text{LEARN-H} \frac{\mathbb{T}, \Gamma, G, \mathcal{R} \quad \Delta := \{\delta \mid \mathcal{T} \in \mathbb{T} \text{ and } \mathcal{T}_{\downarrow\Gamma}, \Gamma, \mathcal{R}, \sigma \rightsquigarrow_G \delta\} \quad \text{for each } \delta \in \Delta: H \Downarrow_\delta \mathbb{T}}{\mathbb{T}, \Gamma, G, \mathcal{R} \rightsquigarrow \forall e_i : \tau_i \in \Gamma. G \rightarrow H}$$

Fig. 6. Formal semantics of SPECY learning procedure (selected rules)

for a term, and a true (\top) or false (\perp) value for a predicate. Additionally, we define an observation δ that maps terms with symbolic events to concrete values, based on a given σ . We use $T \Downarrow_\delta v$ and $P \Downarrow_\delta b$ to denote that the Dynamic Learner Interface evaluates T, P to v, b , respectively, under observation δ .

Learning ϕ_\forall formulas. The semantics of learning a ϕ_\forall formula is encoded in Rules (2) and (4) in Fig. 6. Operationally, SPECY computes a *set of instantiations*, denoted Σ , as follows:

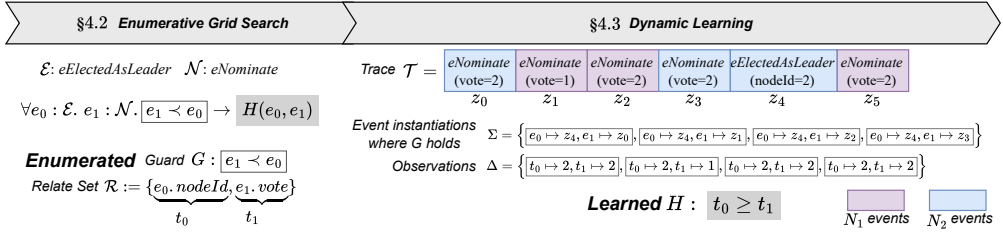
$$\Sigma = \bigcup_{\mathcal{T} \in \mathbb{T}} \{\sigma \mid \mathcal{T}_{\downarrow\Gamma}, \Gamma \triangleright \sigma \text{ and } \mathcal{T}_{\downarrow\Gamma}, \Gamma, \sigma \models G\}$$

Intuitively, Σ represents *all* instantiations σ of symbolic events by concrete events in the given traces, such that G evaluates to true under σ . Now, if there is a predicate P_H , such that $\text{EVAL}(\sigma, P_H)$ is true under all $\sigma \in \Sigma$, then it holds under the guard G and can therefore be included in H in the target formula.

To implement the above operations, for each $\sigma \in \Sigma$, SPECY computes an observation $\delta = \{t \mapsto \text{EVAL}(\sigma, t) \mid t \in \mathcal{R}\}$, which maps terms in the enumerated Relate Set \mathcal{R} to their evaluations under σ . The semantics of an observation δ is shown in the CHECK-G rule (Rule 2), where $\mathcal{T}, \Gamma, \mathcal{R}, \sigma \rightsquigarrow_G \delta$ captures that δ is an observation of terms in the given Relate Set \mathcal{R} where G holds. The application of Rule 2 to each $\sigma \in \Sigma$ results in a *set* of observations Δ computed as follows:

$$\Delta = \{\{t \mapsto \text{EVAL}(\sigma, t) \mid t \in \mathcal{R}\} \mid \sigma \in \Sigma\}$$

Then, as shown in the LEARN-H rule (Rule 4), SPECY leverages a dynamic learner to learn predicates in H that evaluate to true in each observation $\delta \in \Delta$. As an example, suppose $\mathcal{R} = \{e_0.a, e_1.b\}$ and SPECY computes the following observations Δ (where $t \mapsto v$ indicates that term t evaluates to value v under some σ): $\Delta = \{\delta_1 = \{e_0.a \mapsto 0, e_1.b \mapsto 2\}, \delta_2 = \{e_0.a \mapsto 2, e_1.b \mapsto 3\}\}$. In

Fig. 7. Running Example: Learning a ϕ_V formula for the RLE protocol.

this example, the dynamic learner can learn predicates $H : e_0.a < e_1.b, e_0.a \geq 0, e_1.b > 0$, as they hold on all observations in Δ , i.e., for each $\delta \in \Delta, H \Downarrow_{\delta} \top$, as required by the LEARN- H rule.

Running Example. Fig. 7 shows the running example when learning a \forall -only formula with the event combination $\vec{e}_i = \langle \mathcal{E}, \mathcal{N} \rangle$. The static Grid Search enumerates $G = e_1 < e_0$ and Relate Set $\mathcal{R} = \{e_0.nodeId, e_1.vote\}$. Using the trace shown in Fig. 7, the dynamic learner interface computes the event instantiations Σ and observations Δ , and the dynamic learner discovers the predicate in H as $e_0.nodeId \geq e_1.vote$. This results in $\forall e_0 : \mathcal{E}, e_1 : \mathcal{N}. e_1 < e_0 \rightarrow e_0.nodeId \geq e_1.vote$ (§4.5).

Learning $\phi_{V\exists}$ formulas. For handling the existential quantifiers, SPECY computes *witnesses* for each observation of terms involving only universally quantified symbolic events, as shown in the CHECK-GW rule (Rule 3) in Fig. 6.

In more detail, SPECY partitions the event typing context Γ into two disjoint contexts Γ_V and Γ_{\exists} , that denote the contexts for universally and existentially quantified symbolic events, respectively. Similarly, \mathcal{R} is partitioned into two disjoint subsets, $\mathcal{R}_{V\exists}$ and \mathcal{R}_V . The partition $\mathcal{R}_{V\exists}$ contains terms t where some symbolic events from Γ_{\exists} appear in t , while \mathcal{R}_V contains terms t such that no symbolic event from Γ_{\exists} appears in t . Then, SPECY computes instantiations for symbolic events in Γ_V and observations for terms in \mathcal{R}_V , similar to Rule 2 for learning \forall -only formulas. For each σ , such that $\mathcal{T}_{\downarrow\Gamma_V}, \Gamma_V, \mathcal{R}_V, \sigma \rightarrow_G \delta$, SPECY computes *witness* instantiations σ' for symbolic events in Γ_{\exists} , such that W holds. That is, $\mathcal{T}_{\downarrow\Gamma}, \Gamma_V \cup \Gamma_{\exists}, \sigma \cup \sigma' \models W$. Since there can be multiple such σ' , the witnesses of the corresponding observation δ is a *set* of observations, shown as $\omega(\delta)$ in the CHECK-GW rule. This results in the set of instantiations Σ and the set of observations Δ computed as follows:

$$\Sigma = \{(\sigma, \sigma' \in \Sigma') \mid \mathcal{T}_{\downarrow\Gamma_V}, \Gamma_V, \sigma \models G \text{ and } \mathcal{T}_{\downarrow\Gamma}, \Gamma_V \cup \Gamma_{\exists}, \sigma \cup \sigma' \models W\}$$

$$\Delta = \{(\delta, \omega(\delta)) \mid \mathcal{T}_{\downarrow\Gamma_V}, \Gamma_V, \mathcal{R}_V, \sigma \rightarrow_G \delta \text{ where } (\sigma, \Sigma') \in \Sigma\}$$

Intuitively, for each observation δ of terms in \mathcal{R}_V where G holds, this procedure gives a set of witnesses $\omega(\delta)$, such that for every $\delta' \in \omega(\delta)$, W holds under $\delta \cup \delta'$. Importantly, if any $\omega(\delta)$ is empty, then the learning procedure immediately returns with no specifications learned for the enumerated G, W , since an empty witness indicates that either (1) there is no valid σ' for existentially quantified events, or (2) W does not hold under any witness.

The learning procedure for $\phi_{V\exists}$ takes all the $(\delta, \omega(\delta))$ pairs from Δ as input to a dynamic learner, which finds predicates H such that for each $\delta, \omega(\delta)$, there is a $\delta' \in \omega(\delta)$ such that $H \Downarrow_{\delta \cup \delta'} \top$. The Set Cardinality constraint $SC(m)$ can be learned from the number of such distinct witnesses δ' in each $\omega(\delta)$, i.e., $|\{\delta' \mid \delta' \in \omega(\delta) \text{ and } H \Downarrow_{\delta \cup \delta'} \top\}|$.

Running Example. 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 $G = \top, W = e_1 < e_0$, and $\mathcal{R} = \{e_0.nodeId, e_1.vote\}$. Using the trace shown in Fig. 7, SPECY computes Δ as shown below:

$$\Delta = \left\{ \left(\{e_0.nodeId \mapsto 2\}, \{ \{e_1.vote \mapsto 2\}, \{e_1.vote \mapsto 1\}, \{e_1.vote \mapsto 2\}, \{e_1.vote \mapsto 2\} \} \right) \right\}$$

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 Δ . This avoids an explicit tracking of disjunctions that would be required to find some witness in each $\omega(\delta)$. In our example, since 2 (from the first component) is a member of the (multi) set $\{2, 1, 2, 2\}$ (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 $e_0.nodeId = e_1.vote$ 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 (3): $\forall e_0 : \mathcal{E}. \exists e_1 : \mathcal{N}. e_1 \prec e_0 \wedge e_0.nodeId = e_1.vote$.

Another example with cardinality constraints is presented in Appendix A, and the complete set of rules for the semantics of the learning procedure is shown in Appendix B.

4.4 Soundness of SPECY Learning Procedure

We present key lemmas and theorems that prove the soundness of SPECY's learning procedure with respect to the semantics (Fig. 5) and given input traces; the proofs are included in Appendix B.

LEMMA 4.1 (COMPLETENESS OF INSTANTIATIONS). *For each trace \mathcal{T} and event typing context Γ*

$$\{\sigma \mid \mathcal{T}, \Gamma \triangleright \sigma\} = \{\sigma' \mid \sigma'(e) = z, z : \tau \in \mathcal{T}, \Gamma(e) = \tau\}$$

The above lemma states the completeness of instantiations for symbolic events e in the formula with concrete events z in the trace \mathcal{T} . It helps establish that the observations and witnesses are computed from a complete set of instantiations of events in a given trace.

LEMMA 4.2 (OBSERVATION LIFTING- T). *Let t be a term and σ be an event instantiation. For every observation δ that maps terms t to $EVAL(\sigma, t)$, if $t \Downarrow_{\delta} v$, then $EVAL(\sigma, t) = v$.*

LEMMA 4.3 (OBSERVATION LIFTING- P). *Let φ be a quantifier-free formula and σ be an event instantiation. For every observation δ that maps terms t to $EVAL(\sigma, t)$, if $\varphi \Downarrow_{\delta} b$, then $EVAL(\sigma, \varphi) = b$.*

The above two lemmas state that one can *lift* observations of terms δ , which maps terms with symbolic events to their evaluations, to the instantiation σ of symbolic events with concrete events, which was used for computing the evaluations of terms. These help establish the premise that the learned predicates H indeed evaluate to true on instantiations of symbolic events.

Finally, we have our soundness theorems stated as follows:

THEOREM 4.4 (SOUNDNESS- \forall). *Given a set of traces \mathbb{T} , event combination \vec{e}_i , enumerated guards G , and relate set \mathcal{R} . If $\mathbb{T}, \vec{e}_i, G, \mathcal{R} \rightsquigarrow \phi$ then $\mathbb{T} \models \phi$.*

THEOREM 4.5 (SOUNDNESS- $\forall\exists$). *Given a set of traces \mathbb{T} , event combinations \vec{e}_i, \vec{e}_j , enumerated guards G , witnesses W , and relate set $\mathcal{R} = \mathcal{R}_{\forall} \cup \mathcal{R}_{\exists}$. If $\mathbb{T}, \vec{e}_i \cup \vec{e}_j, G, W, \mathcal{R}_{\forall} \cup \mathcal{R}_{\exists} \rightsquigarrow \phi$, then $\mathbb{T} \models \phi$.*

4.5 Formula Assembly and Pruning

After the dynamic learner has learned predicates for H , SPECY uses sanitization steps based on event payload types to remove those that are irrelevant or do not express relationships between quantified events. For example, comparing two HTTP status codes using \leq may get discovered as a predicate by the dynamic learner, but this is unlikely to be useful. After such predicates are removed (by a sanitization pass that prunes predicates that are not allowed in the type-based grammar), 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 .

```

1 spec Monitor observes  $\mathcal{E}, \mathcal{N}$  {
2   # filtered trace
3   var e0s: seq[Event];
4   var e1s: seq[Event];
5   var indexof: map[Event, int];
6   # definitions of G, H
7   fun G(e0:  $\mathcal{E}$ , e1:  $\mathcal{N}$ ): bool {
8     return indexof[e1] < indexof[e0];
9   }
10  fun H(e0:  $\mathcal{E}$ , e1:  $\mathcal{N}$ ): bool {
11    return e0.nodeId >= e1.vote;
12  }
13  start state Observe {
14    on eElectedAsLeader do (z:  $\mathcal{E}$ ) {
15      indexof[z] = sizeof(e0s#e1s);
16      e0s.append(z);
17      foreach (z1 in e1s) {
18        assert  $\neg G(z, z1) \vee H(z, z1)$ ;
19      }
20    }
21  }
22  # handler of eNominate is similar
23  on eNominate ... do ...
24 }

```

Fig. 8. Snippet of runtime monitor for specification $\forall e_0 : \mathcal{E}. e_1 : \mathcal{N}. e_1 < e_0 \rightarrow e_0.\text{nodeId} \geq e_1.\text{vote}$

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.

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 ϕ_{\forall} formulas with the same quantified events. For instance, consider two formulas $I_1 = \forall \vec{e}_i : \vec{\tau}_i. G(\vec{e}_i) \rightarrow \exists \vec{e}_j : \vec{\tau}_j. H(\vec{e}_i, \vec{e}_j)$, $I_2 = \forall \vec{e}_i : \vec{\tau}_i. G'(\vec{e}_i) \rightarrow \exists \vec{e}_j : \vec{\tau}_j. H'(\vec{e}_i, \vec{e}_j)$. If $G' \Rightarrow G$ and $H \Rightarrow H'$, then I_1 subsumes I_2 . In this case, SPECY keeps I_1 and prunes I_2 . We use syntactic checking first because it is very fast and does not require an SMT solver (e.g., Z3 [12]), although it can miss some semantic implications due to overapproximation in the abstraction of predicates.

Pruning by semantic checking. To detect implications missed by syntactic checking, SPECY leverages Z3 [12] to check implications $G \Rightarrow H$, and then uses them for detecting tautologies and subsumptions. This supports Basic Predicates (P) using their standard interpretations and treats user-defined predicates/functions as uninterpreted. The latter may miss an implication (that would otherwise require additional user-provided axioms, which we avoid).

Pruning symmetric formulas. Two ϕ_{\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 $e_0.b > e_1.a$). For example, $\forall e_0, e_1 : \tau. e_0.x_1 < e_1.x_2 \rightarrow e_0.y_1 < e_1.y_2$ is equivalent by symmetry to $\forall e_0, e_1 : \tau. e_0.x_2 > e_1.x_1 \rightarrow e_0.y_2 > e_1.y_1$, and SPECY only keeps one of them.

4.6 Generating Runtime Monitors

SPECY automatically translates each learned specification after pruning into a *specification monitor* (SM). Fig. 8 shows a snippet of the SM generated for the specification in our running example (shown in Fig. 7). The SM is implemented as a P state machine with event handlers for each quantified event type \mathcal{E}, \mathcal{N} (e.g., line 14). It maintains *filtered traces* that store only events relevant to the specification. Each handler checks that appending an incoming event to the trace maintains the specification invariant. For instance, when an *eElectedAsLeader* event z is observed, the SM first appends z to the type-compatible filtered trace (line 16). It then checks all instantiations of the quantified events e_0, e_1 by binding e_0 to the incoming event z and e_1 to each previously observed \mathcal{N} event, according to the semantics in Fig. 5 (lines 17–19). If any assertion fails, the SM emits an abort signal. Full listings for example monitors are shown in Appendix E.

Table 3. Comparing a set of golden ($\varphi_U \in S_U$) and SPECY-learned ($\varphi_L \in S_L$) specifications

Relationship	What it indicates	Development actions
$S_U \subset S_L$	SPECY learns missing specifications	Add missing specifications to tests
$\varphi_L \Rightarrow \varphi_U$	Inadequate behavioral coverage in testing	Improve tests to relax/weaken φ_L
Other cases	Inadequate traces or search space Potential bug in the implementation	Diversify traces with more tests Rigorously debug with $S_U \setminus S_L$

5 Impact of Learned Specifications in Practice

Specifications learned by SPECY provide immediate benefits in different stages of development, from formal reasoning to runtime monitoring and revealing gaps in behavioral test coverage.

5.1 Using Learned Specifications for Formal Reasoning

SPECY aids formal reasoning about the system by learning a diverse set of specifications, including both *safety properties* (§6.3) (for model checking) and *inductive invariants* (§6.4) that enable a deductive verifier [42] to prove the learned safety properties. Previously, both the safety properties and the inductive invariants needed to be provided manually by a user, often difficult for developers.

5.2 Using Learned Specifications for Runtime Monitoring

SPECY automatically generates runtime monitors from learned specifications that observe messages exchanged in the distributed system, and hence can be easily applied to traces from production systems that log these messages. Runtime monitoring with learned specifications enables rigorous validation of production systems against formal models, ensuring that implementations adhere to the design specifications despite potential optimizations. This is particularly effective because, in industry practice, while formal models and production systems may have vastly different state spaces, their message interfaces are often similar. In a tool like Dinv, considerable effort is required to translate state-based specifications into local runtime monitors [19]. In contrast, SPECY provides an automated and straight-forward translation, thanks to message interfaces. This helps to *bridge the specification gap* between formal models and system implementations via largely automated learning and translation by SPECY. We demonstrate this in our evaluations (§6.5.1). Similar benefits can be realized for comparing systems before and after refactoring for performance optimization.

5.3 Using Learned Specifications to Identify Gaps in Test Coverage

SPECY can also assist developers in understanding system behaviors and in identifying behavioral coverage gaps in testing, by comparing developer-written specifications (S_U) with SPECY-learned specifications (S_L). We denote individual specifications in S_U and S_L as φ_U and φ_L , respectively, and consider various relationships of interest and what they indicate, as summarized in Table 3.

- $S_U \subset S_L$: **Discovering missing specifications.** When SPECY learns specifications beyond those originally written by developers, the additional specifications in $S_L \setminus S_U$ often capture properties overlooked previously during system design but critical for correctness, which can then be added to existing tests. Our evaluations demonstrate this in one of our proprietary systems where S_U was available (§6.5.2).
- $\varphi_L \Rightarrow \varphi_U$: **Revealing gaps in behavioral test coverage.** This scenario represents semantic entailment, where SPECY learns a stronger specification than that written by developers over the *same* event combination. SPECY can learn stronger specifications than desired due to either (1) weaker guards G , or (2) stronger witness and hypotheses $W \wedge H$. Both cases reveal coverage gaps in system behavioral testing among the traces provided to SPECY as inputs. Weaker guards in φ_L indicate that system behaviors are inadequately explored, failing to trigger stronger control

conditions. Stronger $W \wedge H$ in φ_L may contain extra predicates in the specifications because test inputs over-constrain system behavior. We demonstrate in our evaluations that SPECY helps identify such coverage gaps in existing tests (§6.5.2).

Other cases. Although we did not observe cases such as $S_L \subset S_U$ or $\varphi_U \Rightarrow \varphi_L$, these scenarios may happen in practice and can provide useful feedback to users about the input traces given to SPECY. In particular, $S_L \subset S_U$ suggests that the input traces or the space explored by the Grid Search may be insufficient, failing to capture behaviors that users consider important. $\varphi_U \Rightarrow \varphi_L$ indicates that the input traces could originate from a bug in the high-level model, and they could violate the stronger user-provided specification φ_U . Other discrepancies, such as $S_L \neq S_U$, may arise from a combination of these factors.

6 Evaluations

Our evaluation addresses the following research questions:

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

RQ2: Do the specifications learned by SPECY help deductive verification, particularly learning inductive invariants? How does it compare with prior work?

RQ3: How does SPECY help developers in practice? In particular, we demonstrate its benefits through: (a) runtime-monitoring, and (b) identifying gaps in behavioral test coverage.

6.1 Implementation

We implemented SPECY as an extension of the P compiler [4, 13], with approximately 7,000 lines of code in C#. We implemented a Dynamic Learner Interface (§4.3) with about 1,400 lines of code in Java to inter-operate with Daikon [15]. SPECY uses the Z3 solver [12] for semantic checking in pruning. As a specialized mode within P, SPECY accepts formal models from P users, autonomously invokes the PChecker tool [4] (which systematically explores different event orderings) to generate event traces, and subsequently presents the learned specifications to the user. For deductive verification, learned specifications are manually translated to PVerifier [42] syntax to prove the safety properties. For runtime monitoring, SPECY automatically generates executable monitors that can be used by PChecker [4] for P models or by existing test infrastructure for production systems.

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 [3, 29], SWISS [21], and I4 [2, 41], and a Firewall model from PVerifier [4, 42]. In addition, we developed P models for 6 foundational protocols: Two-Phase Commit (2PC) [20], Ring Leader Election [10], Paxos [34], Chain Replication [53], Vertical Paxos [36] and Raft [44]. 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 transactional service that combines multi-version concurrency control [7] with two-phase commit [20] to guarantee atomicity and snapshot isolation. The formal models

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 $\forall\exists$ is needed). UG2 and UG3 show the number of lines of auxiliary code that provide user guidance (complete listings shown in Appendix C). The last column shows the end-to-end run time of SPECY (in seconds).

Benchmark	LoC	S_{goals}	Description of Specifications	UG2	UG3	Time (s)
Ring Leader Election [10]	47	3 ✓	Unique leader, Leader Highest ID, Leader Voted	0	0	352
Consensus [†] [29]	61	1 ✓	Safety (Unique Decision)	0	0	925
2PC [4]	238	2 ✓	Atomicity ($\forall\exists_{SC}$), Safety (Commit or Abort)	0	0	3935
Sharded KV ^{†*} [29]	48	1 ✓	Safety (Unique Shard Owner)	0	0	481
Paxos [34]	164	2 ✓	Safety (Unique Decision), B3(\mathcal{B}) [†] [34]	0	0	1155
Distributed Lock [†] [41]	73	1 ✓	Safety (Mutual Exclusion)	0	0	902
Vertical Paxos [36]	241	2 ✓	Safety (Unique Decision), B3(\mathcal{B}) [†] [36]	0	0	2589
Firewall [42]	80	1 ✓	Whitelisted Safety ($\forall\exists$)	0	0	753
Lock Server [†] [41]	102	1 ✓	Safety (Mutual Exclusion)	0	0	620
Chain Replication [53]	452	6 ✓	Update Propagation, ⁺ Inprocess Requests, ⁺ Linearizability ($\forall\exists$) ^{†#}	44	12	984
Raft [†] [44]	1191	5 ✓	Election Safety, State Machine Safety, Log Matching ⁺ Leader Append-Only, Leader Completeness ⁺	54	6	4223
GlobalClock	177	3 ✓	Clock Monotonicity, Real-time ordering [#]	0	0	805
DBLeaderElection	1178	5 ✓	Monotonic Commits, Unique Leader, Read Consistency ^{†#}	0	0	697
MVCC-2PC	1135	7 ✓	Atomicity ($\forall\exists_{SC}$) [#] , Snapshot Isolation ($\forall\exists$) ^{†#}	0	66	2090

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

[†] P model translated from an IVy model [†] 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 [#] Entailed by a subset of the learned specifications

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 learned important specifications that were missing in the handwritten specifications. We also conducted a case study with the DBLeaderElection benchmark, for learning specifications from its production traces that were available.

Setup. The experiments were conducted on a machine with two AMD EPYC 7R13 48-core CPUs and 1.5TB of memory. For each benchmark, input traces are generated using PChecker [13] with 3–5 different system configurations, i.e., number of processes/inputs or failures. For example, for Paxos, we varied the number 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 capability of SPECY to *effectively learn essential correctness properties* is a fundamental requirement for its practicality as a specification learning framework. To evaluate this, we created a challenge set comprising 43 protocol specifications, which we designate as *goal specifications* (S_{goals}) that should be learned by SPECY. These specifications are extracted from seminal papers that introduced the protocols in our benchmarks [10, 29, 34, 36, 41, 44, 46, 53], and for industrial case studies, these include specifications authored by the developers themselves. We note that 5 of these protocol specifications require $\forall\exists$ -quantifiers, including 2 with quorum constraints – the full set of first-order logic (FOL) formulas for these specifications are provided in Appendix D. To determine

whether a goal specification is learned by SPECY, we use light-weight semantic entailment checking (part of our pruning procedures).

Table 4 summarizes the results, where the LoC column reports the number of lines of code in each P model (roughly indicating protocol complexity), the S_{goals} column lists the number of goal specifications in each benchmark, and the last column reports the end-to-end execution time of SPECY (in seconds). *Notably, all 43 goal specifications are learned by SPECY for these benchmarks, with 28 (65%) of these learned automatically without any user intervention.* The remaining 15 required some degree of user guidance; footnotes in the table denote these instances, and the columns UG2 and UG3 report the number of lines of code added for user-guidance.

Learning from production traces. We also conducted a case study using traces collected from a production deployment of the DBLeaderElection benchmark. We used 114 traces collected over a one hour period, from 109 nodes distributed across 3 data centers. Due to administrative constraints, these traces contain only leadership-related events; however, these events use the same message interface as our P model. Using these traces, SPECY successfully learned two critical properties of the leader election part: the *Unique Leader* specification and the monotonicity invariant of generation numbers in leadership relinquishment events.

6.3.1 Discussion on User Guidance. Among the challenge set, 15 of 43 specifications require user guidance for SPECY to learn them successfully. SPECY supports three types of user guidance (§3.3) to handle cases where automatic learning falls short. All code added for user guidance in these cases is listed in Appendix C.

UG1: Event combination specification requires *no additional code* and addresses cases where SPECY’s causality-based heuristics miss important event combinations. Users may leverage their insights of the protocol to specify crucial event combinations in the form of configuration inputs to SPECY. For example, in Chain Replication, we provided a read-write response event combination to learn 2 formulas that are part of the *Linearizability* specification. In total, we used UG1 to learn 7 challenge specifications.

UG2: Custom predicates are needed when specifications involve domain-specific operations on custom data types that generic dynamic learners cannot handle. To provide UG2, users may write the custom predicates in code as P functions, to guide SPECY to learn specifications with the added predicates. For instance, SPECY learned Raft’s Log Matching property after adding a custom predicate \sqsubseteq , where $l_1 \sqsubseteq l_2$ indicates that log sequences l_1 and l_2 match up to a certain index, i.e., $l_1 \sqsubseteq l_2 \iff \forall i. l_1[i] = l_2[i] \rightarrow \forall j \in [0, i). l_1[j] = l_2[j]$.

UG3: Exposing states involves extending the message interface by adding code or instrumenting the system to selectively expose relevant local states as event payloads. We found that exposing local states that reflect *shared data*, such as commit logs or transaction records, is particularly effective because distributed protocols typically maintain consistency over such shared state. In general, providing UG3 is harder than the previous two cases, since it involves selecting which shared states to expose, followed by adding code and/or instrumentation of the message interface. For example, for learning MVCC-2PC’s Snapshot Isolation we added code to expose transaction commit records, while for learning Chain Replication, we added code to expose the commit log.

6.3.2 Effectiveness of Pruning Specifications after Dynamic Learning. We evaluated the effectiveness of the pruning steps on all benchmarks, shown in Table 5. The dynamic learner Daikon [15] can learn a large number of specifications (shown in the S_{raw} column). SPECY first applies pruning by syntactic checking and then by semantic checking. The number of specifications remaining after each stage is shown in columns S_{syn} and S_{sem} , respectively. Our results show an average (geo-mean) reduction by $14.8\times$ (RR column). Notably, in all benchmarks, the final number of

Table 5. Number of specifications after each pruning step

Benchmark	S_{raw}	S_{syn}	S_{sem}	RR	$S_{\text{falsified}}$
Ring Leader Election	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	72	58	18×	10
GlobalClock	368	37	37	9×	0
DBLeaderElection	577	50	47	12×	9
MVCC-2PC	2017	143	89	22×	24

Table 6. Learning inductive invariants using SPECY (RQ2). We use a subset of benchmarks with proofs in PVerifer [42]. I_e and I_s columns show the numbers of necessary event-based and state-based inductive invariants in the proofs, respectively. Time column shows the PVerifier time (ms) on *all* learned specifications (left) vs. only on $I_e \cup I_s$ (right).

Benchmark	I_e	I_s	Time (ms)
Ring Leader Election*	1	2	1181 / 153
Consensus*	1	5	135 / 113
Distributed Lock*	4	4	137 / 101
Lock Server*	4	4	107 / 98
Firewall	1	0	99 / 87
Sharded KV*	1	1	853 / 95

learned specifications (S_{sem}) is *less than a hundred*, while including all those in our challenge set. Additionally, SPECY can automatically invoke PChecker [4] to check the learned specifications on additional traces (outside the input set of traces given to SPECY), and prune the falsified ones. Our strategy in using a large number of diverse traces (over multiple configurations) is geared toward learning generalizable specifications that hold also for unseen traces. In our experiments, only 9% of learned specifications were falsified using PChecker (shown in the $S_{\text{falsified}}$ column), without eliminating any among the goal specifications.

6.4 RQ2: Learning Inductive Invariants

To evaluate SPECY’s utility for downstream verification tasks, we experimented with benchmarks previously verified using PVerifer [42]—a framework for deductive verification of P programs. Our evaluation focused specifically on the *inductive invariants* required by PVerifer proofs that were previously provided manually by users (formulas are provided in Appendix D). These inductive invariants fall into two categories: event-based and state-based, with the latter requiring additional user guidance (UG3) in the form of 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, respectively, required for correctness proofs. The total number of learned specifications in each benchmark is the same as S_{sem} column in Table 5. The Time column reports the time (in milliseconds) to prove the safety property using PVerifer [42]: the left value shows the verification time using *all* learned specifications, while the right value shows the verification time using only the necessary inductive invariants ($I_e \cup I_s$). With user-defined code (for benchmarks marked with asterisks *), SPECY *successfully learns all inductive invariants that were previously required to be provided manually*. Without such user guidance, SPECY still learns all event-based invariants (I_e) but fails to learn the state-based ones (I_s). In particular, the verifier can find a proof without any additional input from the user. This is unlike a recent work Basilisk [63], where a user-written proof of inductiveness is also required. This demonstrates SPECY’s capability in aiding downstream verification tasks by *significantly reducing developers’ burden for finding inductive invariants*, which is notoriously difficult.

Impact on verifier performance. Note that SPECY may learn additional specifications and invariants beyond those in the challenge set and those required to prove the specifications. While providing alternative views of model behaviors, these “unnecessary” invariants may slow down the verifier

(Time column in Table 6). Techniques such as iterative Houdini-style approaches [14, 17, 47] can be used to identify a maximal inductive set of invariants, but they are outside the scope of this paper.

Discussion on comparison with related tools: DuoAI and SWISS. At a high level, SPECY and DuoAI [60]/SWISS [21] target related but slightly different problems: SPECY aims to learn protocol specifications from event traces, whereas DuoAI and SWISS aim to learn inductive invariants for proofs, *given* the protocol specifications and a model. DuoAI and SWISS are state-of-the-art tools that can effectively find proofs for the protocols shown in Table 6 within 8 and 20 seconds, respectively [21, 60]. Note that this includes the search for inductive invariants, where their grammars constrain the search space. Both tools leverage the Ivy verifier [46] to prune invalid candidates, and the resulting proofs are formally verified. However, to use these tools, developers must first invest significant intellectual effort to develop decidable models of their protocols, which may not be straight-forward [45]. In contrast, SPECY does not impose this requirement: it learns specifications from event traces of executable models or system implementations. This allows SPECY to learn specifications for complex protocols like Vertical Paxos [36] and Raft [44], which have not been reported by either DuoAI or SWISS (due to the limitations of EPR or lack of support for specific protocol features). While these complex protocols have not (yet) been verified by PVerifier either, the specifications learned by SPECY can be used for runtime monitoring, to identify gaps in test coverage, and to aid developers' understanding. Thus, when a complex protocol is beyond the scope of existing verifiers such as Ivy, or specifications are not yet known, SPECY provides a practical alternative.

The choice of learning from execution traces rather than a model with a verifier comes at a cost: SPECY searches a more complex space during learning (involving interpreted predicates and data types) and analyzes a large set of traces (for effective learning). Therefore, as expected, it incurs a longer run time than DuoAI and SWISS for learning invariants (time reported in Table 4). The lack of a verifier also means that, unlike DuoAI and SWISS, SPECY alone does not produce formally verified proofs, and depends on a downstream verifier [42] that supports verification of event-based specifications. This verifier is quite fast though, completing verification in seconds in each case (Time column in Table 6), since the inductive invariants learned by SPECY are sufficient for a proof.

6.5 RQ3: Benefits in Development Practices

We demonstrate how learned specifications can be utilized to improve development practices.

6.5.1 Benefits in Runtime Monitoring. We evaluated support for runtime monitoring by automatically translating specifications learned by SPECY from input testing traces, into executable runtime monitors for production traces. For the DBLeaderElection case study, we applied 5 such runtime monitors on production traces collected from a live deployment, analyzing 11,045 leadership-related events (from the same traces described in §6.3). Monitoring revealed that 1 out of 5 leadership-related specifications learned by SPECY on the P model was violated in production traces. Importantly, we discovered that these violations did not indicate bugs, but rather highlighted that the learned specification was *overly restrictive* for the production environment. The violated specification stated that leader generation numbers must be identical across *all* leadership relinquishment events – this holds in the P model, but is *too strong* for production scenarios.

This case study demonstrates two key benefits of SPECY discussed earlier (§5): (1) many specifications learned from the formal model were successfully validated by runtime monitoring on the production system, and (2) the violations provided valuable feedback to developers about the discrepancies between formal model and real-world system behavior. Notably, these benefits are enabled by our design choice in SPECY to learn specifications on *events*, rather than on global states. The message interfaces are similar in the formal P model and the production system, which is often

the case in industry practice – this allows the event-based specifications learned from one to be applied to the other for the purpose of cross-validation.

6.5.2 Identifying Gaps in Specifications and Test Coverage. SPECY learned three critical specifications that were overlooked by developers in the proprietary MVCC-2PC protocol: (1) only the leader shard server can initiate a commit of transaction; (2) each transaction can only be prepared at most once by each shard server, and (3) transaction status on participant servers should be consistent with its status on the leader shard server. These specifications are essential for correctness – if violated, any shard server could initiate commits and prepare requests multiple times, resulting in committed transactions without consensus. Team members familiar with the protocol confirmed that these were absent from the hand-written specifications, but are required for correct operation.

In the DBLeaderElection monitoring case study on the production system (described earlier, §6.5.1), a specification learned by SPECY on the formal model was stronger than desired (it was violated on the actual production behavior). This observation informed the development team that they could consider additional behaviors (similar to production runs) during model testing. Similarly, in MVCC-2PC, SPECY learned a specification with a *stronger* $W \wedge H$. In the input test traces to SPECY, the shard servers never failed to commit transactions, leading SPECY to learn specifications with additional predicates $e_0.status = SHARD_OK$ for all shard commit events e_0 . Upon review by developers, this prompted them to improve their test inputs to include shard server failure cases in behavioral tests.

6.6 Limitations

While SPECY is highly effective, it has several limitations. First, SPECY does not guarantee completeness in discovering all correct specifications for a given protocol, and may discover specifications that are falsifiable on traces it has not seen. This limitation comes from two sources. Due to tradeoffs in expressiveness, as discussed in §3.1, SPECY does not currently support learning specifications with a $\exists\forall$ quantifier pattern. Supporting this and other patterns (e.g., $\forall\exists\forall$) in SPECY is possible by extending the Dynamic Learner Interface (§4.3) to compute the event instantiations, witnesses, and observations according to the semantics of the specifications over traces (§3.1). The second source is due to traces provided as input to SPECY. Similar to any dynamic approach, the quality and diversity of input traces are critical for SPECY, where *inadequate* trace coverage can lead to incomplete or incorrect specification learning, as discussed earlier (§5). Second, while SPECY is designed to significantly reduce user burden, some domain-specific properties might require user guidance (UG1–3), as demonstrated in our evaluations (§6.3). Providing user guidance usually involves domain knowledge about the protocol and code, for adding code or instrumentation. Third, SPECY uses pruning procedures based on syntactic and light-weight semantic checking, but avoids more expensive procedures for checking equivalence or entailment. This may result in too many learned formulas. In future work, we would like to investigate additional pruning techniques [25], and add ranking of learned formulas to prioritize usage in downstream tasks.

7 Related Work

Verifier-aided invariant synthesis. SWISS [21] and DuoAI [60] are state-of-the-art tools for learning inductive invariants (including $\forall\exists$ -quantifiers) to formally prove given safety properties using verifiers [46, 54]. As discussed earlier (§6.4), they do not target learning correctness specifications (that are not inductive). Various earlier efforts [18, 28, 29, 41, 47, 61] are similar, but some do not support nested existential quantifiers.

Kondo [62] and Basilisk [63] synthesize Hoare-style specifications using static analyses of Dafny models [38]. Their specification formula patterns capture many (state-based) invariants useful for

safety proofs, and they describe a taxonomy of invariants (classified as regular, protocol, provenance invariants) to guide the automated search for inductive invariants. However, these patterns do not capture many safety properties supported by SPECY, e.g., those with temporal relationships. Plus, the reusability of their specifications in runtime verification on production systems is unclear.

Data-driven invariant learning. Dinv [19] and LIDO [56] are leading efforts that learn safety properties from state traces, i.e., traces of global system states. Although these approaches scale better than verifier-based approaches, they have significant limitations in supporting expressive specifications, e.g., with $\forall\exists$ -quantifiers or quorum constraints, requiring user intervention for manual generalization to quantified specifications (as discussed earlier, §1). A recent work, FORCE [59], encodes formula enumeration using state traces as an answer set programming problem, aiming to replace DuoAI’s enumeration procedure. Although more efficient than DuoAI, formulas enumerated by FORCE and DuoAI offer the same level of expressiveness, and neither supports temporal relations between message events or specifications for runtime monitoring.

Other earlier efforts [8, 30, 40, 57, 58] focused on learning *property patterns* over execution traces or message sequence charts. Specifically, Perracotta [58] uses pre-defined property templates to recognize patterns, Beschastnikh et al. [8] focuses on temporal relationships between events, and Kumar et al. [30] uses a set of regular expression templates to find message sequence patterns. Although their temporal patterns can be viewed in terms of quantifiers and precedence over the *occurrence* of events, they do not support expressing any relationships between *event payloads*. 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 [39, 49, 50, 55] use interactive theorem-provers such as Rocq and Iris [27, 52] to formulate and find correctness proofs. Semi-automated efforts [22, 23, 37, 42, 45, 51, 62] leverage user-provided annotations to formulate verification tasks handled by automated verifiers such as Ivy [46, 54], Dafny [38], or Uclid [31]. Both sets of efforts have been successfully demonstrated on complex distributed systems, but require significantly higher manual effort than SPECY. Model checking and automatic state exploration have also been used to find bugs in distributed systems [13, 26, 33], 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 framework that learns correctness specifications from event traces of complex distributed systems. It uses a specialized grammar to learn specifications with quantified events and relationships between their payloads, and provides support for $\forall\exists$ -quantifiers and quorum constraints. SPECY uses a novel learning procedure in which a static enumerative search is combined effectively with dynamic learning on event traces, followed by pruning procedures. 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 learn inductive invariants that benefit a downstream verifier, and its learned specifications can improve development practices by automatic translation to runtime monitors on production systems and by identifying gaps in behavioral test coverage.

Acknowledgements

We thank the OOPSLA reviewers for their constructive feedback, which substantially improved the paper. We also thank Abtin Molavi for early contributions and discussions on SPECY and Han Xu

for helpful discussions on the notation in the formalizations. This work was supported in part by NSF grants 2422053, CNS-2319442, and an Amazon Research Award.

Data-Availability Statement

The artifact of SPECY evaluated in §6 is available on Zenodo [24]. The most up-to-date versions are available on GitHub [1, 4].

References

- [1] [n. d.]. GitHub - AD1024/PInfer-OOPSLA-Artifact: Artifact of "Specy: Learning Specifications for Distributed Systems from Event Traces" — github.com. <https://github.com/AD1024/PInfer-OOPSLA-Artifact>. [Accessed 22-02-2026].
- [2] [n. d.]. 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].
- [3] [n. d.]. GitHub - jrkoenig/folseparators: First Order Logic Separators — github.com. <https://github.com/jrkoenig/folseparators>. [Accessed 11-04-2025].
- [4] [n. d.]. GitHub - p-org/P: The P programming language. — github.com. <https://github.com/p-org/P>. [Accessed 18-03-2025].
- [5] Wei An, Xiao Bi, Guanting Chen, Shanhuang Chen, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Wenjun Gao, Kang Guan, Jianzhong Guo, Yongqiang Guo, Zhe Fu, Ying He, Panpan Huang, Jiashi Li, Wenfeng Liang, Xiaodong Liu, Xin Liu, Yiyuan Liu, Yuxuan Liu, Shanghao Lu, Xuan Lu, Xiaotao Nie, Tian Pei, Junjie Qiu, Hui Qu, Zehui Ren, Zhangli Sha, Xuecheng Su, Xiaowen Sun, Yixuan Tan, Minghui Tang, Shiyu Wang, Yaohui Wang, Yongji Wang, Ziwei Xie, Yiliang Xiong, Yanhong Xu, Shengfeng Ye, Shuiping Yu, Yukun Zha, Liyue Zhang, Haowei Zhang, Mingchuan Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, and Yuheng Zou. 2024. Fire-Flyer AI-HPC: A Cost-Effective Software-Hardware Co-Design for Deep Learning. arXiv:2408.14158 [cs.DC] <https://arxiv.org/abs/2408.14158>
- [6] Robert Beers. 2008. Pre-RTL formal verification: An Intel experience. In *2008 45th ACM/IEEE Design Automation Conference*. 806–811. doi:10.1145/1391469.1391675
- [7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *SIGMOD Rec.* 24, 2 (May 1995), 1–10. doi:10.1145/568271.223785
- [8] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, Arvind Krishnamurthy, and Thomas E. Anderson. 2011. 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* (Cascais, Portugal) (SLAML '11). Association for Computing Machinery, New York, NY, USA, Article 3, 10 pages. doi:10.1145/2038633.2038636
- [9] Marc Brooker and Ankush Desai. 2025. Systems Correctness Practices at AWS: Leveraging Formal and Semi-formal Methods. *Queue* 22, 6 (Feb. 2025), 79–96. doi:10.1145/3712057
- [10] Ernest Chang and Rosemary Roberts. 1979. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM* 22, 5 (May 1979), 281–283. doi:10.1145/359104.359108
- [11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*.
- [12] Leonardo De Moura and Nikolaj Bjørner. 2008. 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* (Budapest, Hungary) (TACAS'08/ETAPS'08). Springer-Verlag, Berlin, Heidelberg, 337–340.
- [13] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) (PLDI '13). Association for Computing Machinery, New York, NY, USA, 321–332. doi:10.1145/2491956.2462184
- [14] Sushant Dinesh, Yongye Zhu, and Christopher W. Fletcher. 2025. H-Houdini: Scalable Invariant Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) (ASPLOS '25). Association for Computing Machinery, New York, NY, USA, 603–618. doi:10.1145/3669940.3707263
- [15] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1–3 (Dec. 2007), 35–45. doi:10.1016/j.scico.2007.01.015

- [16] George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes* 22, 4 (July 1997), 74–80. doi:10.1145/263244.263267
- [17] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. 2001. Annotation inference for modular checkers. *Inf. Process. Lett.* 77, 2–4 (Feb. 2001), 97–108. doi:10.1016/S0020-0190(00)00196-4
- [18] Aman Goel and Karem Sakallah. 2021. 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*. Springer-Verlag, Berlin, Heidelberg, 131–150. doi:10.1007/978-3-030-76384-8_9
- [19] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering (Gothenburg, Sweden) (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1149–1159. doi:10.1145/3180155.3180199
- [20] Jim Gray. 1988. *The transaction concept: virtues and limitations*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 140–150.
- [21] Travis Hance, Marijn Heule, Ruben Martins, and Bryan Parno. 2021. 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)*. USENIX Association, 115–131. <https://www.usenix.org/conference/nsdi21/presentation/hance>
- [22] Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. 2023. Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 911–929. <https://www.usenix.org/conference/osdi23/presentation/hance>
- [23] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (Monterey, California) (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/2815400.2815428
- [24] Deyuan (Mike) He, Ankush Desai, Jagarapu Aishwarya, Terry Doug, Sharad Malik, and Aarti Gupta. 2026. *Artifact for OOPSLA'26: Specv: Learning Specifications for Distributed Systems from Event Traces*. doi:10.5281/zenodo.18452033
- [25] Mike He, Zhendong Ang, Ankush Desai, and Aarti Gupta. 2025. Ranking Formal Specifications using LLMs. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages (Singapore, Singapore) (LMPL '25)*. Association for Computing Machinery, New York, NY, USA, 51–56. doi:10.1145/3759425.3763386
- [26] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (May 1997), 279–295. doi:10.1109/32.588521
- [27] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. 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 (Mumbai, India) (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. doi:10.1145/2676726.2676980
- [28] Aleksandr Karbyshev, Nikolaj Børner, Shachar Itzhaky, Noam Rinetzkzy, and Sharon Shoham. 2017. Property-Directed Inference of Universal Invariants or Proving Their Absence. *J. ACM* 64, 1, Article 7 (March 2017), 33 pages. doi:10.1145/3022187
- [29] Jason R. Koenig, Oded Padon, Neil Immerman, and Alex Aiken. 2020. First-order quantified separators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (London, UK) (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 703–717. doi:10.1145/3385412.3386018
- [30] Sandeep Kumar, Siau-Cheng Khoo, Abhik Roychoudhury, and David Lo. 2012. Inferring class level specifications for distributed systems. In *Proceedings of the 34th International Conference on Software Engineering (Zurich, Switzerland) (ICSE '12)*. IEEE Press, 914–924.
- [31] Shuvendu K. Lahiri and Sanjit A. Seshia. 2004. The UCLID Decision Procedure. In *Computer Aided Verification*, Rajeev Alur and Doron A. Peled (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 475–478.
- [32] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. doi:10.1145/359545.359563
- [33] Leslie Lamport. 1994. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 872–923. doi:10.1145/177492.177726
- [34] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. doi:10.1145/279227.279229
- [35] Leslie Lamport. 2019. Industrial Use of TLA+ — lamport.azurewebsites.net. <https://lamport.azurewebsites.net/tla/industrial-use.html>. [Accessed 10-03-2025].
- [36] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (Calgary, AB, Canada) (PODC '09)*. Association for Computing Machinery, New York, NY, USA, 312–313. doi:10.1145/1582716.1582783

- [37] 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. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) (SOSP '24). Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952
- [38] K. Rustan M. Leino. 2010. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) (LPAR'10). Springer-Verlag, Berlin, Heidelberg, 348–370.
- [39] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) (POPL '16). Association for Computing Machinery, New York, NY, USA, 357–370. doi:10.1145/2837614.2837622
- [40] Chang Lou, Yuzhuo Jing, and Peng Huang. 2022. Demystifying and Checking Silent Semantic Violations in Large Distributed Systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 91–107. <https://www.usenix.org/conference/osdi22/presentation/lou-demystifying>
- [41] Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) (SOSP '19). Association for Computing Machinery, New York, NY, USA, 370–384. doi:10.1145/3341301.3359651
- [42] Federico Mora, Ankush Desai, Elizabeth Polgreen, and Sanjit A. Seshia. 2023. Message Chains for Distributed System Verification. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 300 (Oct. 2023), 27 pages. doi:10.1145/3622876
- [43] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardouff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (March 2015), 66–73. doi:10.1145/2699417
- [44] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>
- [45] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 108 (Oct. 2017), 31 pages. doi:10.1145/3140568
- [46] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. *SIGPLAN Not.* 51, 6 (June 2016), 614–630. doi:10.1145/2980983.2908118
- [47] Oded Padon, James R. Wilcox, Jason R. Koenig, Kenneth L. McMillan, and Alex Aiken. 2022. Induction duality: primal-dual search for invariants. *Proc. ACM Program. Lang.* 6, POPL, Article 50 (Jan. 2022), 29 pages. doi:10.1145/3498712
- [48] Koushik Sen. 2007. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) (ASE '07). Association for Computing Machinery, New York, NY, USA, 571–572. doi:10.1145/1321631.1321746
- [49] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (Dec. 2017), 30 pages. doi:10.1145/3158116
- [50] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) (SOSP '23). Association for Computing Machinery, New York, NY, USA, 113–129. doi:10.1145/3600006.3613172
- [51] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. *SIGPLAN Not.* 51, 1 (Jan. 2016), 256–270. doi:10.1145/2914770.2837655
- [52] The Coq Development Team. 2024. *The Coq Proof Assistant* – <https://doi.org/10.5281/zenodo.14542673>. doi:10.5281/zenodo.14542673
- [53] Robbert van Renesse and Fred B. Schneider. 2004. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) (OSDI'04). USENIX Association, USA, 7.
- [54] James R. Wilcox, Yotam M. Y. Feldman, Oded Padon, and Sharon Shoham. 2024. 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* (Montreal, QC, Canada). Springer-Verlag, Berlin, Heidelberg, 71–85. doi:10.1007/978-3-031-65630-9_4
- [55] James R. Wilcox, Doug Woos, Pavel Panckekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) (PLDI '15). Association

- for Computing Machinery, New York, NY, USA, 357–368. doi:10.1145/2737924.2737958
- [56] Yuan Xia, Deepayan Sur, Aabha Shailesh Pingle, Jyotirmoy V. Deshmukh, Mukund Raghothaman, and Srivatsan Ravi. 2025. Discovering Likely Invariants for Distributed Systems Through Runtime Monitoring and Learning. In *Verification, Model Checking, and Abstract Interpretation*, Krishna Shankaranarayanan, Sriram Shankaranarayanan, and Ashutosh Trivedi (Eds.). Springer Nature Switzerland, Cham, 3–25.
- [57] Maysam Yabandeh, Marco Canini, Dejan Kostic, and Abhishek Anand. 2011. Finding Almost-Invariants in Distributed Systems. In *Reliable Distributed Systems, IEEE Symposium on*. IEEE Computer Society, Los Alamitos, CA, USA, 177–182. doi:10.1109/SRDS.2011.29
- [58] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering (Shanghai, China) (ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 282–291. doi:10.1145/1134285.1134325
- [59] Ziyi Yang, George Pirlea, and Ilya Sergey. [n. d.]. Inductive First-Order Formula Synthesis by ASP: A Case Study in Invariant Inference. ([n. d.]).
- [60] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 485–501. <https://www.usenix.org/conference/osdi22/presentation/yao>
- [61] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 405–421. <https://www.usenix.org/conference/osdi21/presentation/yao>
- [62] Tony Nuda Zhang, Travis Hance, Manos Kapritsos, Tej Chajed, and Bryan Parno. 2024. 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)*. USENIX Association, Santa Clara, CA, 837–853. <https://www.usenix.org/conference/osdi24/presentation/zhang-nuda>
- [63] Tony Nuda Zhang, Keshav Singh, Tej Chajed, Manos Kapritsos, and Bryan Parno. 2025. Basilisk: Using Provenance Invariants to Automate Proofs of Undecidable Protocols. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 1–17.
- [64] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (Sept. 2022), 36 pages. doi:10.1145/3512345

Received 2025-10-10; accepted 2026-02-17