Ranking Formal Specifications using LLMs

Mike He
Princeton University
Princeton, USA
mikehe@princeton.edu

Zhendong Ang
National University of Singapore
Singapore, Singapore
zhendong.ang@u.nus.edu

Ankush Desai
Amazon Web Services
Seattle, USA
ankushpd@amazon.com

Aarti Gupta
Princeton University
Princeton, USA
aartig@cs.princeton.edu

Abstract

Formal specifications are essential for reasoning about the correctness of complex systems. While recent advances have explored automatically learning such specifications, the challenge of distinguishing meaningful, non-trivial specifications from a vast and noisy pool of learned candidates remains largely open. In this position paper, we present an approach for specification ranking, aimed at identifying the most critical specifications that contribute to overall system correctness. To this end, we develop a four-metric rating framework that quantifies the importance of a specification. Our approach leverages the reasoning capabilities of Large Language Models to rank specifications from a set of automatically learned candidates. We evaluate the proposed method on a set of specifications inferred for 11 open-source and 3 proprietary distributed system benchmarks, demonstrating its effectiveness in ranking critical specifications.

CCS Concepts: • **Theory of computation** → **Program specifications**; • **Software and its engineering**;

Keywords: Ranking specifications, Agentic workflows

ACM Reference Format:
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 (LMPL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3759425.3763386

1 Introduction

The formal reasoning of complex systems is critically dependent on the correctness specifications. Traditionally, such specifications have been manually authored by experts – a process that is both a significant bottleneck and highly resource-intensive, as it demands deep domain expertise and considerable engineering effort. To alleviate this challenge, previous research has explored methods for automatically learning specifications from system behaviors, for instance, analyzing execution traces of the system [8, 9, 12, 13, 16, 22].

While automated specification learning reduces the burden on developers, its effectiveness on large-scale systems is often limited by the sheer volume of learned specifications. Data-driven approaches typically rely on grammar-based search [4, 9, 21], which produces an exponentially large space and overwhelmingly many trivial specifications. For example, without manual guidance, Dinv [9] infers nearly one million properties for the Raft consensus protocol [2, 18]. More recent tools attempt to address this challenge: the Daikon-based learner [4] restricts grammar and employs logic-based pruning to eliminate redundant specifications, and further incorporates a model-checking-based fuzzer [5] to falsify incorrect ones using a P [7] model. However, even with these optimizations, the output for complex systems still consists of hundreds of candidate specifications. As a result, the challenge shifts from learning specifications to identifying the critical ones that are most relevant for downstream tasks such as verification. We refer to this new challenge as the **specification ranking** problem.

1.1 Challenges and Our Solution

We address the specification ranking problem by leveraging the reasoning capabilities of Large Language Models (LLMs). In doing so, we identify two central challenges: **Challenge (C1): Lack of a formal notion of criticalness.** The *criticalness* of specifications is not formally defined and varies by context. For example, in consensus protocols such as Paxos [15], ensuring the uniqueness of decisions is critical, whereas in database systems, atomicity and consistency typically take precedence. **Challenge (C2): Dependence on system-specific expertise.** Effective classification of specifications requires knowledge of system design and intended behaviors. This expertise may exceed a typical user's understanding or necessitate specialized tools. To address **(C1)**, we introduce a four-metric rating framework that evaluates a specification in four dimensions: generalizability, criticality,

LMPL '25, Singapore, Singapore
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2148-9/25/10
https://doi.org/10.1145/3759425.3763386

Figure 1. Simple Client-Server protocol in P

```
 1  # Event declarations with payload types
 2  event Request: (clt: Client, reqId: int);
 3  event Response: (reqId: int);
 4  # Client state machine
 5  machine Client {
 6    var server: Server;
 7    var rid: int;
 8    start state SendRequest {
 9      entry (srv: Server) {
10        server = srv;
11        rid = randomId();
12        send server, Request, (clt=this, reqId=rid);
13        goto WaitResponse;
14      }
15    }
16    state WaitResponse {
17      on Response do (payload: (reqId: int)) {
18        assert(payload.reqId == this.rid);
19      }
20    }
21  }
22  # Server state machine
23  machine Server {
24    start state Serving {
25      on Request do (req: (clt: Client, reqId: int)) {
26        send req.clt, Response, (reqId=req.reqId);
27      }
28    }
29  }
```

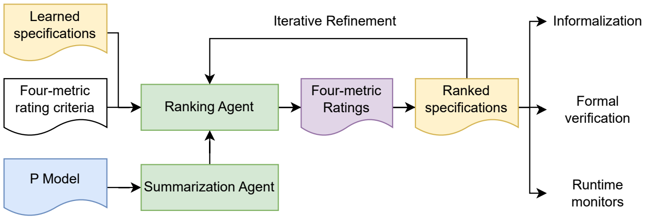
distinguishability, and visibility. These metrics capture complementary aspects of the importance of specifications, ranging from broad applicability to user-facing impact, and can be extended or customized for different systems and domains. To address **(C2)**, we employ LLM agents that analyze the implementations of the system and apply our rating framework to evaluate the learned specifications. We evaluated our approach on distributed system benchmarks by ranking specifications generated by a state-of-the-art dynamic learner [4]. Compared to a baseline method that ranks by predicate frequency, our LLM-based approach achieves significantly higher alignment with developer-identified specifications. Specifically, in 9 out of 14 benchmarks, all developer-identified *target* specifications appear in the top-10 ranked candidates, where as the baseline achieves this only once. Furthermore, our approach identifies previously overlooked but critical properties in proprietary systems, demonstrating its potential for aiding practical verification workflows. Our prototype is publicly available.[1] While this paper primarily uses distributed systems as case studies, the proposed approach is general and can be applied to other domains.

2 Background

2.1 P language and modeling framework

P [5, 7] is a state-machine-based programming language designed for formal modeling and analysis of distributed systems. A P program comprises state machines that communicate asynchronously through *events* carrying typed data values. The machines execute concurrently, receiving and sending events while updating their local states.

Figure 2. LLM-based specification ranking workflow.



Example: Simple Client-Server Protocol. To illustrate the main concepts of a P program, consider a simple client-server protocol in which a client sends a request to the server, which then responds. The P model is shown in Figure 1. The `Request` and `Response` declarations (lines 2–3) define the events and their payload types. For example, the `Request` payload includes a reference to the client and a request ID. Each state machine (lines 5, 23) specifies associated states (lines 8, 16, 24) and local variables (lines 6, 7). A state may include an entry function (line 9) that executes upon entering that state. After executing this entry function, the machine dequeues events from its buffer and executes the corresponding *event handler*. For instance, in the `WaitResponse` state, the machine handles `Response` events (line 17). Machines communicate via `send` operations (lines 12, 26) and change states using `goto` (line 13). The explorer tool PChecker [5] executes P models and records *event traces* capturing all message exchanges between machines. These event traces serve as input to specification learning tool.

2.2 Learning specifications from event traces

We employ a dynamic learner [4] that generates specifications for distributed systems from event traces. Given event traces recorded by PChecker, this tool automatically discovers specifications expressed as first-order logic formulas over events, characterizing system behaviors.

Specification Learning Example. For the client-server example, the tool generates specifications such as the following. We denote a *Request* as $\mathcal{R}$ and a *Response* as $\mathcal{S}$.

$$\forall e_0 : \mathcal{S}.\, \exists e_1 : \mathcal{R}.\, e_1 \prec e_0 \land e_0.reqId = e_1.reqId \tag{1}$$

$$\forall e_0, e_1 : \mathcal{S}.\, e_0.reqId = e_1.reqId \longrightarrow e_0 = e_1 \tag{2}$$

$$\forall e_0 : \mathcal{R}, e_1 : \mathcal{S}.\, e_0.reqId = e_1.reqId \longrightarrow e_0 \prec e_1 \tag{3}$$

These specifications capture the following properties:
Eqn 1: Every response corresponds to a preceding request, where $\prec$ denotes the happens-before [14] relation.
Eqn 2: Responses are unique per request ID.
Eqn 3: Requests precede their corresponding responses.

[1] p-org/P

Table 1. Example ratings in the four-metric framework. For each example, the first line shows the specification formula, and the second line shows the score and justification. $eWriteSuccess$ and $eWriteFailure$ are important user-visible events.

Metrics	Rating examples in System Prompt
$G(\phi)$	$\forall e_0 : eAbort.\, \forall e_1 : eCommit.\, e_0.id \neq e_1.id$ 1: Universally true - abort and commit never occur for the same transaction in ANY execution.
	$\forall e_0 : eCommit.\, \exists e_1 : ePrepareSuccess.\, e_1 \prec e_0 \land e_0.id = e_1.id \land e_0.voter \neq e_1.voter$ 0.5: Generally true but may not hold in single-node deployments.
	$\forall e_0 : ePrepareReq.\, \exists e_1 : ePrepareFailure.\, e_0 \prec e_1$ 0: Not generalizable to traces with no failed transaction.
$C(\phi)$	$\forall e_0 : eAbort.\, \forall e_1 : eCommit.\, e_0.id \neq e_1.id$ 1: Violation leads to data corruption requiring manual rollback.
	$\forall e_0 : eCommit.\, \forall e_1 : eCommit.\, e_0.id = e_1.id \rightarrow e_0 = e_1$ 0.5: Moderately critical - duplicate commits cause inconsistency but may be detectable and recoverable.
	$\forall e_0 : ePrepareSuccess.\, \exists e_1 : ePrepareReq.\, e_1 \prec e_0$ 0: Non-critical: the opposite ($e_0 \prec e_1$) may also be true for multiple transactions.
$D(\phi)$	$\forall e_0 : eCommit.\, \exists_n e_1 : ePrepareSuccess.\, e_1 \prec e_0 \land e_0.id = e_1.id$ 1: Rejects all executions with incomplete prepare successes.
	$\forall e_0 : eCommit.\, \exists e_1 : ePrepareReq.\, e_1 \prec e_0 \land e_0.id = e_1.id$ 0.6: Catches commits without prepare requests but misses commits with failed prepares.
	$\forall e_0 : eAbort.\, \exists e_1 : ePrepareFailure.\, e_0.id = e_1.id$ 0.3: Not strong ePrepareFailureugh to reject executions where the prepare failure happens after the eAbort.
$V(\phi)$	$\forall e_0 : eWriteSuccess.\, \forall e_1 : eWriteFailure.\, e_0.id \neq e_1.id$ 1: Operator immediately notices contradictory transaction outcomes.
	$\forall e_0 : eAbort.\, \exists e_1 : eWriteFailure.\, e_0.id = e_1.id$ 0.5: Abort and Commit can trigger user-visible events.
	$\forall e_0 : eCommit.\, \exists e_1 : ePrepareReq.\, e_1 \prec e_0 \land e_0.id = e_1.id$ 0.1: Internal protocol ordering not directly visible to application users.

For more complex systems with richer event interactions, the tool can learn hundreds of specifications from event traces. Our goal is to rank these learned specifications to identify the most critical subset for system correctness.

3 Ranking with LLMs

The specification ranking problem presents two primary challenges: first, the notion of a specification's importance lacks a quantitative definition; second, understanding and evaluating specifications requires domain expertise in the protocol design. To address the first challenge, we propose a *four-metric rating framework* that quantifies specification's contribution to system correctness. For the second challenge, we design a ranking workflow that integrates our rating framework with the reasoning capabilities of Large Language Models (LLMs) to *rank* specifications accordingly.

Figure 2 illustrates our overall workflow. Initially, *a summarization agent* analyzes the P model to extract event definitions, state machine descriptions, and event flows between machines. Subsequently, *a ranking agent* receives the learned specifications alongside a detailed explanation of our four-metric rating framework with illustrative examples, and the context analysis of the P model. Using this information, the ranking agent evaluates each specification according to our framework and produces a ranking based on the computed ratings. This ranking process iterates until user-defined criteria are met (e.g., a target number of specifications or minimum average rating). During each refinement iteration, the ranking agent receives a subset of k_n specifications from the previous iteration and is tasked with producing a ranked subset of size k_{n+1} where $0 < k_{n+1} < k_n$. This staged reduction is valuable when the initial set is large, mitigating inaccuracies that can arise from aggressive one-shot filtering.

3.1 Four-Metric Rating Framework

Our rating framework evaluates each specification ϕ along four dimensions, producing scores in $[0.0, 1.0]$ for each metric. The four metrics are derived from intuitions that good specifications should be *generalizable* across different configurable parameters, capturing *correctness* properties and rejecting *undesired* behaviors of the system. For example, different system configurations may involve different number of nodes, network topologies, or workload patterns (e.g., read-heavy, write-heavy, etc.) Table 1 presents examples of metric scores for specifications learned in the Two-Phase

Commit protocol [10], including the scores, justifications, and prompt examples (full prompts are publicly available[2]).

Generalization Score $G(\phi)$**.** The generalization score $G(\phi)$ reflects the likelihood that a specification represents a system *invariant* that holds across all valid executions, independent of system configuration or implementation. It distinguishes between specifications that capture general correctness versus those that overfit to specific configurations. Examples with different scores are shown in Table 1 ($G(\phi)$ row).

Criticality Score $C(\phi)$**.** The criticality score $C(\phi)$ evaluates the severity of consequences when a specification is violated, measuring the *blast radius* of potential failures and their impact on system correctness and recoverability. This metric captures whether the violation of a specification would cascade into system-wide failures or remain localized. Specifications with high criticality scores protect against violations that result in unrecoverable system states or compromise essential safety properties. The $C(\phi)$ row of Table 1 shows examples with various scores for this metric.

Distinguishability Score $D(\phi)$**.** The distinguishability score $D(\phi)$ measures how effectively a specification serves as a separator that differentiates between correct and incorrect system behaviors. Specifications with high distinguishability should be *strong* enough to exclude erroneous behaviors, while weak enough to apply to all correct behaviors. Examples are shown in the $D(\phi)$ row of Table 1.

Visibility Score $V(\phi)$**.** The visibility score $V(\phi)$ measures how directly a specification's properties are observable by end users or system operators. We focus on specifications defined over *events* observed during system execution, where events can be either observable to the user of the system (e.g., responses) or internal to the system (e.g., cluster reconfiguration). High visibility score indicates that the violation of the specification is "closer" to the interface between the system and its users, making it easier to detect and debug. Some examples are shown in the $V(\phi)$ row of Table 1.

Overall Rating and Ranking. Given the metric scores $G(\phi)$, $C(\phi)$, $D(\phi)$, and $V(\phi)$ for a specification ϕ, the overall score $S(\phi)$ is computed as:

$$S(\phi) = \lambda_1 \cdot \sqrt{G(\phi) \cdot C(\phi)} + \lambda_2 \cdot D(\phi) + \lambda_3 \cdot V(\phi)$$

where λ_i are hyper-parameters such that $\sum_i \lambda_i = 1$. Here, $\sqrt{G(\phi) \cdot C(\phi)}$ is a *quality* term that ensures that specifications with high generalization but low criticality (or vice versa) do not dominate the ranking. The distinguishability and visibility scores are added linearly, allowing them to contribute to the overall score without overpowering the quality term (controlled via λ_2, λ_3). The Ranking agent is prompted to follow the rating framework and provide the top-k rated specifications. The k value can be decreased iteratively until the user is satisfied with the final result.

Figure 3. Preliminary results: The first column shows the benchmark name and the number of learned specifications as input for ranking. The next five columns show the fraction of target specifications included in top-k after ranking. The last column shows the cost for LLM invocations. The first 11 rows are benchmarks with open-source protocols, and the next 3 rows are proprietary benchmarks.

	k=10	k=20	k=30	k=40	k=50	Cost
2PC (46)	2/2	2/2	2/2	2/2	2/2	$0.07
Chain (33)	3/5	5/5	5/5	5/5	5/5	$0.15
Raft (58)	3/5	5/5	5/5	5/5	5/5	$0.12
Toy Consensus (28)	1/1	1/1	1/1	1/1	1/1	$0.07
Distributed Lock (76)	1/1	1/1	1/1	1/1	1/1	$0.20
Whitelist (40)	1/1	1/1	1/1	1/1	1/1	$0.07
Lock Server (35)	1/1	1/1	1/1	1/1	1/1	$0.08
Paxos (46)	2/2	2/2	2/2	2/2	2/2	$0.11
Ring Leader (27)	1/1	1/1	1/1	1/1	1/1	$0.08
Sharded KV (19)	1/1	1/1	1/1	1/1	1/1	$0.06
Vertical Paxos (94)	2/2	2/2	2/2	2/2	2/2	$0.21
ClockBound (37)	2/3	3/3	3/3	3/3	3/3	$0.09
DynamoDB-LE (54)	1/5	1/5	3/5	3/5	5/5	$0.15
2PC-CC (89)	2/7	3/7	5/7	7/7	7/7	$0.26
Overall	**23/37**	**29/37**	**33/37**	**35/37**	**37/37**	**$1.71**
Coverage (%)	**62.2%**	**78.4%**	**89.2%**	**94.6%**	**100.0%**	**-**

4 Experiments

We have conducted preliminary experiments to evaluate the effectiveness of our approach. We use the state-of-the-art Claude Sonnet 4 [1] LLM for creating the Summarization and Ranking agents. We applied our approach to 11 well-known protocols (including Raft [18], Paxos [15], and Chain Replication [20], etc.) and 3 proprietary protocols written in P [5] language, including: a clock service developed at AWS that allows for the generation and comparison of bounded timestamps (ClockBound), a leader election protocol of the DynamoDB service (DynamoDB-LE), and an internal Two-Phase Commit protocol that guarantees Snapshot Isolation [6] (2PC-CC). We constructed the target specification set (37 in total) identified by prior literature [3, 15, 17, 18] for open-source protocols and specifications composed by development teams for proprietary protocols. Note that our proprietary benchmarks are *not publicly accessible*; therefore the LLM we use in our evaluations does not have knowledge about their implementations or their specifications. We used a recent dynamic learner [4] to learn specifications as inputs to the ranking process. We set the weights to

[2] p-org/P/tree/experimental/pinfer/Src/PInfer/Scripts/Prompts

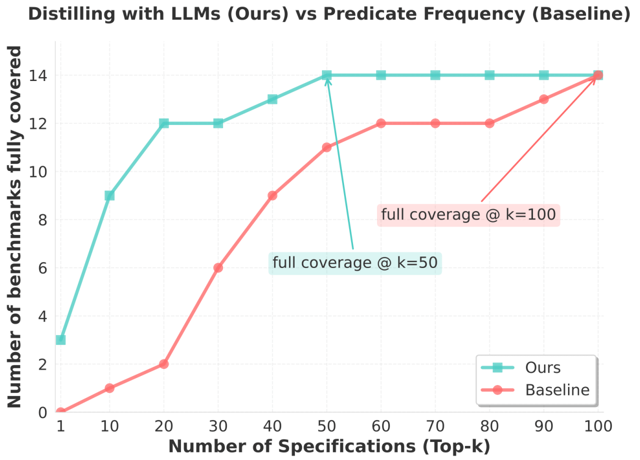


Figure 4. Number of benchmarks whose target specifications are all found within top-k using different approaches.

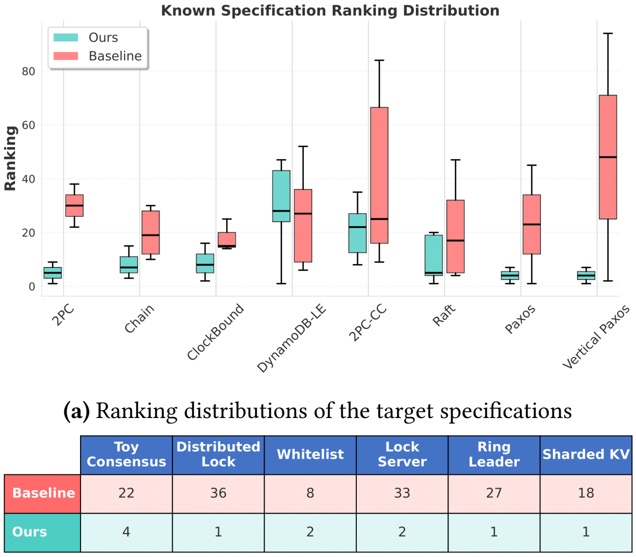
$\lambda_1 = 0.8, \lambda_2 = 0.1$ and $\lambda_3 = 0.1$ to compute overall scores to favor specifications with higher quality terms. For iterative refinement, we use the number of ranked specifications as the termination condition and prompt the Ranking Agent to output specifications ranked in the top 50%.

Figure 3 shows our results. Notably, all target specifications in 9 of the 14 benchmarks are covered within the top-10 ranked candidates, with negligible LLM cost. We also examined high-ranked specifications outside the target set, which fell into two categories: (1) redundant formulas expressing the same property as a target specification, and (2) critical properties missing from the target set. While the former adds little value due to their redundancy, the latter is particularly important as it highlights critical specifications the developers may have overlooked. For instance, a top-ranked specification for the proprietary 2PC-CC protocol states that users should not observe inconsistent transaction status between the leader server and shard servers. While this specification was not in the target set identified by the developers, it represents a crucial correctness property. This demonstrates a key benefit of our approach: it can help developers discover important specifications that they may have missed during development. Upon validation, developers can use them to strengthen protocol specifications.

4.1 Comparisons with baseline approach

We compared our ranking workflow with a baseline approach based on predicate frequency in the learned specifications. Specifically, the baseline approach first computes f_p, the frequency of each predicate p that appears in the set of learned specifications. Then, it assigns each predicate a weight $w_p = f_p^{-1}$. Now, the weight of a specification W_ϕ is computed as $W_\phi = \sqrt{\prod_{p_i \in \phi} w_{p_i}}$. Finally, the learned specifications are ranked by W_p in descending order.

We compared the number of benchmarks whose target specifications are fully covered when taking top-k ranked specifications by each approach, for different values of k. Results are shown in Figure 4. Notably, our approach can fully cover target specifications of 12 of the 14 benchmarks by taking top-20 ranked candidates, whereas the baseline only covers 2. Figure 5 shows in detail how our approach ranks target specifications compared to the baseline. For benchmarks with multiple target specifications, Figure 5a shows ranking distributions reported by each approach, where lower rankings indicate better performance. For benchmarks with only a single target specification, Figure 5b shows the specific ranking position of the target specification with each approach. In all benchmarks, our approach achieves full coverage at a smaller percentile and ranked the target specifications closer to the top than the baseline approach, demonstrating its effectiveness in identifying important properties.



(a) Ranking distributions of the target specifications

	Toy Consensus	Distributed Lock	Whitelist	Lock Server	Ring Leader	Sharded KV
Baseline	22	36	8	33	27	18
Ours	4	1	2	2	1	1

(b) Rankings of the target specifications

Figure 5. Comparison of rankings on individual benchmarks (lower is better): 5a shows the distributions of rankings for benchmarks with *multiple target specifications*; 5b shows the rankings for benchmarks with *a single target specification.*

5 Future work

We plan to incorporate our workflow into a broader automated verification framework. In this framework, given a system model and a set of traces, specification learning tools automatically generate a set of candidate specifications. Then, the ranking process identifies critical specifications that should be further verified using downstream verifiers (e.g., PVerifier [5, 17] for P models, or other domain-specific verifiers). Our rating framework is general and extensible, and we can incorporate additional metrics such as *inductiveness* of invariant specifications [11, 19] to facilitate the verification process.

References

[1] [n. d.]. Claude Sonnet 4 — anthropic.com. https://www.anthropic.com/claude/sonnet. [Accessed 07-07-2025].

[2] [n. d.]. GitHub - etcd-io/raft: Raft library for maintaining a replicated state machine — github.com. https://github.com/etcd-io/raft. [Accessed 19-08-2025].

[3] [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].

[4] [n. d.]. GitHub - p-org/P at experimental/pinfer — github.com. https://github.com/p-org/P/tree/experimental/pinfer. [Accessed 07-07-2025].

[5] [n. d.]. GitHub - p-org/P: The P programming language. — github.com. https://github.com/p-org/P. [Accessed 18-03-2025].

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA) *(SIGMOD '95)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

[7] 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. https://doi.org/10.1145/2491956.2462184

[8] 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. https://doi.org/10.1016/j.scico.2007.01.015

[9] 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. https://doi.org/10.1145/3180155.3180199

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

[11] 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

[12] Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. 2005. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *Proceedings. 42nd Design Automation Conference, 2005.* 775–778. https://doi.org/10.1109/DAC.2005.193920

[13] Markus Kusano, Arijit Chattopadhyay, and Chao Wang. 2015. Dynamic Generation of Likely Invariants for Multithreaded Programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 835–846. https://doi.org/10.1109/ICSE.2015.95

[14] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[15] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[16] 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

[17] 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. https://doi.org/10.1145/3622876

[18] 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

[19] 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. https://doi.org/10.1145/2980983.2908118

[20] 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.

[21] 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 Sankaranarayanan, and Ashutosh Trivedi (Eds.). Springer Nature Switzerland, Cham, 3–25.

[22] 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. https://doi.org/10.1109/SRDS.2011.29

Received 2025-07-07; accepted 2025-08-08