Specifying and Testing Distributed Protocols with Action Temporal Logic

JOSÉ JOÃO FERREIRA, Instituto Superior Técnico, Portugal

Distributed systems form the backbone of modern infrastructures such as financial services, cloud platforms, and blockchain networks. Ensuring the correctness of their coordination and fault-tolerant protocols is challenging, as real implementations often diverge from formally verified designs, leading to subtle but critical faults. Offline monitoring can verify protocol implementations by analyzing execution logs against formal specifications. Existing tools like TLA+ and Alloy face scalability issues, and no formal language currently captures temporal and data dependencies in distributed protocols efficiently for monitoring. This work addresses these challenges by introducing a new framework for testing distributed protocols, comprising a tailored temporal logic for specification, Action Temporal Logic (ACTL), which integrates the interval-based reasoning of ATL with the quantification and expressiveness of FOLTL. ACTL provides an intuitive, action-centered syntax for specifying temporal, causal, and data-dependent relationships between protocol operations, demonstrated through examples from DHTs. The framework also includes ACTLChecker, an efficient monitoring tool that parses formal specifications and execution logs to verify whether a trace satisfies a given ACTL formula, using a decision algorithm with optimized preprocessing for scalability. We evaluate our approach using execution logs from OpenChord, an industrial implementation of the Chord protocol, a DHT. ACTLChecker detects protocol violations and efficiently verifies compliant logs, improving verification time and scalability over existing Alloy-based approaches.

CCS Concepts: • **Software and its engineering → Software verification and validation**; • **Computing methodologies → Distributed algorithms**; • **Theory of computation → Temporal logic**.

Additional Key Words and Phrases: Offline Monitoring, Distributed Protocols, Two-Phase Commit, Distributed Hash Tables, Allen Temporal Logic, First-Order Linear Temporal Logic

1 Introduction

Today, modern software increasingly depends on distributed systems, which underpin critical infrastructures such as cloud platforms, financial systems, and blockchains. These systems must operate reliably and continuously, as failures can have severe consequences. Distributed protocols — including Distributed Hash Tables (DHTs), and consensus algorithms [1] — coordinate nodes to ensure communication correctness, consistency, and fault tolerance.

Verifying the correctness of distributed protocols is essential. Although algorithms can be formally proven correct, real implementations often deviate from their specifications, introducing subtle but critical bugs. Formal verification of production code remains difficult and time-consuming, limiting guarantees mostly to design-level correctness.

Several consensus implementations, including Raft [2], Tendermint [3], and Gasper [4], have exhibited severe correctness flaws [5–8], demonstrating the gap between theoretical proofs and deployed systems.

Validation approaches vary widely:

(a) *Deductive verification* with tools like Rocq, or TLAPS provides strong proofs but lacks scalability.
(b) *Model checking* systematically explores system behaviors but suffers from state-space explosion.
(c) *Fuzzing* can uncover unexpected issues yet provides no completeness guarantees.
(d) *Monitoring* checks system executions against specifications and is practical but detects violations post-execution.

This work focuses on *offline monitoring*, which analyzes complete traces after execution, enabling deterministic and exhaustive property validation.

Monitoring distributed protocols is challenging due to timing, causality, and data dependencies. Constructing reliable logs that correctly order events across nodes is essential but difficult due to asynchrony and failures.

Existing tools like Alloy and TLA+ face scalability and expressiveness issues for large traces. Alloy, in particular, struggles with dynamic behaviors and large scopes. Our goal is to achieve more efficient and scalable monitoring than prior Alloy-based approaches such as [9]. Our main contributions are:

A. **Action Temporal Logic (ACTL): a new action-based logic.** We design ACTL, combining interval and point-based reasoning to express temporal, causal, and data dependencies efficiently. ACTL balances expressiveness and tractability through a formal syntax, semantics, and a decision algorithm for trace validation.
B. **Formal specification of protocol properties.** Using ACTL, we formalize and analyze correctness properties of DHT, showcasing how temporal logics describe distributed behaviors.
C. **ACTLChecker: an efficient monitoring tool.** Implemented in Python, ACTLChecker parses specifications and execution logs, builds internal structures, and validates traces via an optimized decision procedure.
D. **Evaluation on OpenChord traces.** We validate DHT properties on OpenChord logs, demonstrating that ACTLChecker works as intended.

Our approach achieves a practical balance between expressiveness and efficiency, enabling scalable trace validation for complex distributed systems — a trade-off not adequately explored in prior work.

1.1 Structure

This document is structured as follows: Section 3 introduces ACTL, detailing its syntax and semantics, presenting a specification of a DHT protocol, as well as the decision procedure, pseudocode, and complexity analysis. Section 4 describes the implementation of ACTLChecker, the tool developed to validate execution traces of distributed protocols against ACTL property specifications. It outlines the formal specification language, the trace processing pipeline, and the structure of the tool's main modules. Section 5 presents the experimental evaluation of our work, assessing ACTLChecker's functionality. Finally, Section 6 concludes the thesis by summarizing

Author's Contact Information: José João Ferreira, josejoaoferreira@tecnico.ulisboa.pt, Instituto Superior Técnico, Lisbon, Portugal.

the main contributions and outlining directions for future research, followed by the references.

1.2 Publications

Some of the preliminary work has already been published in the short paper listed below. That work specifies properties of the DHT distributed protocols in First-Order Linear Temporal Logic (FOLTL). Generally, it explores the use of FOLTL for property specification and concludes that, while it is a powerful formalism, it may not be particularly practical. Consequently, it would be preferable to represent these properties using an alternative logic with syntactic variations that better reflect the nature of operations that span time intervals, receive inputs, and produce outputs. This insight served as a strong motivation for the development of our logic, ACTL. For more details, please refer to the paper.

J. J. Ferreira, N. Policarpo, J. F. Santos, A. Cunha, and A. Gianola, "First-Order Linear Temporal Logic for Testing Distributed Protocols," in *Short Paper Proceedings of the 7th International Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis, OVERLAY 2025, hosted by the 28th European Conference on Artificial Intelligence, ECAI 2025.* [Online]. Available: https://overlay.uniud.it/workshop/2025/papers/ferreira-etal.pdf

2 Related Work

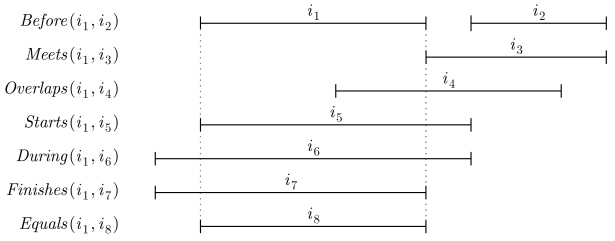
This section reviews research on Allen Temporal Logic (ATL) and FOLTL, their applications, limitations, and tool support, as well as related work on specifying and testing distributed systems.

2.1 Time Reasoning Logics

Several logics enable reasoning about temporal operations and their interrelations, such as ATL [10], LTL [11], FOLTL [12], Temporal Logic of Actions (TLA) [13], and Event Calculus (EC) [14]. We focus on ATL and FOLTL for their suitability in specifying and testing complex systems.

2.1.1 Allen Temporal Logic. Allen Temporal Logic (ATL) [10; 15], or Allen's Interval Algebra, is an interval-based temporal logic for reasoning about relative temporal information [16], e.g., that interval i_1 occurs before or during i_2. ATL focuses on relations between intervals rather than exact time points ($i = [t_1, t_2]$), over discrete or dense time. Seven basic relations — *Before*, *Meets*, *Overlaps*, *Starts*, *During*, *Finishes*, *Equals* — define ATL, with inverses and complex relations derivable. Figure 1 summarizes these predicates.

Fig. 1. Illustration of the interval predicates of ATL relations.



Reasoning is often formulated as a Constraint Satisfaction Problem (CSP), solved via constraint propagation, underpinning Natural Language Processing (NLP) and scheduling systems [17–19]. Applications include modeling temporal relations in clinical events [20] and video analysis [21].

Research explores tractable ATL fragments [22]. Early attempts to verify ATL encoded it into LTL [23], but limitations included discrete-time assumptions and nested operators. Later, encodings into FOLTL [9] and Answer Set Programming (ASP) with difference constraints [24] enabled practical, scalable reasoning.

2.1.2 First-Order Linear Temporal Logic. FOLTL [12] extends First-Order Logic (FOL) with temporal operators, reasoning about how object properties evolve over linear time. Common methods include automata-theoretic approaches, tableau reasoning, and SAT/SMT-based model checking [25; 26], supporting both runtime verification and theorem proving [27].

Applications span monitoring and verification: MonPoly tracks policy and security logs [28], JavaMOP monitors Java programs for runtime defects [29], and linDMT analyzes data-aware processes [30]. Full FOLTL is highly undecidable [31], but practical fragments exist for online monitoring, implemented in tools like BLACK [32] and Alloy [33].

2.2 Specifying and Testing Distributed Protocols

Specification-based testing is widely used to detect violations in distributed protocols, employing temporal logics, process calculi, automata, or state machines. Formal languages like TLA+ [34] model expected behaviors and verify traces from instrumented applications using the TLC model checker. TLA+ has been applied to prevent bugs in AWS protocols [35] and detect violations in executions [36].

Alloy [33] has been used to specify the Chord protocol [37], with the Alloy Analyzer finding counterexamples [38], and later validated larger OpenChord traces using ATL fragments [9].

Model-based trace checking instruments applications to log executions and verifies Linear Temporal Logic (LTL) specifications, e.g., by translating logs into Promela models for Spin [39; 40]. Comparative studies of Alloy and Spin provide guidance on selecting formal methods for protocol verification [41]. Pastry, a well-known DHT protocol, was tested by Lu et al. [42] using a TLA+ model, revealing that concurrent joins could break correctness; they proposed restricting nodes to handle one join at a time, later formally proven by Azmy et al. [43].

Alloy has been used to model Chord, uncovering violations in ring-maintenance invariants and proposing corrections, including a global invariant for eventual network convergence [38; 44]. These examples highlight the need for systematic, trace-based testing of DHTs.

Policarpo et al. [9] introduced an implementation-independent axiomatization of DHTs using an ATL-based approach, with a semi-automatic testing pipeline validating value, key, and structural properties against OpenChord execution logs via Alloy-encoded FOLTL. While effective for small logs (up to 250 entries), Alloy does not scale well for larger traces due to nested quantifiers and memory limits. To address this, we propose a dedicated tool designed for efficient, scalable trace monitoring of distributed protocols.

3 Action Temporal Logic

We introduce Action Temporal Logic (ACTL), combining ATL's interval reasoning with FOLTL's first-order expressiveness and adding actions.

3.1 Syntax and Semantics

3.1.1 Actions. Actions allow specifying system properties by combining operations with ATL relations while abstracting from implementation details. Each action represents a system operation with input parameters, output results, and a time interval of execution. All actions are assumed total, meaning they terminate and produce complete outputs.

An *action* is a formal representation of an operation, and it is denoted as:

$$a(x_1, \ldots, x_n) \overset{i}{\rightsquigarrow} (y_1, \ldots, y_m)$$

where a is the action name, $x_1, \ldots, x_n$ are inputs, $y_1, \ldots, y_m$ are outputs, and i is the execution interval.

This concept was inspired by the formalization of DHTs proposed by Policarpo et al. [9].

3.1.2 Syntax. Let $\mathcal{V}$ be the set of finite *domain variable* symbols, $\mathcal{I}$ the set of time *interval variable* symbols, $\mathcal{A}$ a set of *action* symbols involving variables, and $\mathcal{R}$ the set of Allen's *relation* symbols between interval variables, where these sets are all finite and mutually disjoint. Formulas of ACTL are denoted by φ, and their syntax is defined as follows:

$$\begin{aligned}\varphi \;::=\; & \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \\ & \mid \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y})\,.\,\varphi \mid \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y})\,.\,\varphi \\ & \mid x_1 = x_2 \mid r(i_1, i_2)\end{aligned}$$

where $a \in \mathcal{A}$ is an action symbol, $x \in \mathcal{V}$ is a domain variable symbol, $\overline{x}$ and $\overline{y}$ are, respectively, finite sequences of domain variable symbols $x_1, \ldots, x_{|x|}$ and $y_1, \ldots, y_{|y|}$ where $x_n, y_m \in \mathcal{V}$ (for any $n \in \{1, \ldots, |x|\}$ and $m \in \{1, \ldots, |y|\}$), $i \in \mathcal{I}$ is an interval variable symbol, and $r \in \mathcal{R}$ is a relation symbol. The syntax of Allen's interval relations is defined as follows:

$$\begin{aligned}r(i_1, i_2) \;::=\; & \mathit{Before}(i_1, i_2) \mid \mathit{Meets}(i_1, i_2) \mid \mathit{Overlaps}(i_1, i_2) \\ & \mid \mathit{Starts}(i_1, i_2) \mid \mathit{During}(i_1, i_2) \mid \mathit{Finishes}(i_1, i_2) \\ & \mid \mathit{Equals}(i_1, i_2)\end{aligned}$$

Variable quantification in ACTL is performed collectively over all input, output, and time parameters associated with a given action a. To enhance clarity and readability, the universal and existential action quantifiers notation is, respectively, interpreted as follows:

$$\begin{aligned}\forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \iff & \textbf{forall } i, x_1, \ldots, x_n, y_1, \ldots, y_m\,. \\ & a(x_1, \ldots, x_n) \overset{i}{\rightsquigarrow} (y_1, \ldots, y_m) \\ \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \iff & \textbf{exists } i, x_1, \ldots, x_n, y_1, \ldots, y_m\,. \\ & a(x_1, \ldots, x_n) \overset{i}{\rightsquigarrow} (y_1, \ldots, y_m)\end{aligned}$$

3.1.3 Semantics. Let σ be the *variable environment*, i.e., the mapping from domain variable symbols $x \in \mathcal{V}$ to values in their corresponding domains D_x, and from interval variable symbols $i \in \mathcal{I}$ to pairs (t_b, t_e):

$$\sigma : (\mathcal{V} \to D) \cup (\mathcal{I} \to \mathbb{N} \times \mathbb{N})$$

Let $\mathcal{E}$ be a set of *event symbols*, and let $\varepsilon \in \mathcal{E}$ denote an individual event. Events are classified into two types, as defined below:

$$\varepsilon \;::=\; B\langle a\rangle_k(\overline{v}) \mid E\langle a\rangle_k(\overline{w})$$

where $k \in \Sigma^*$ is an identifier used to distinguish between multiple occurrences of actions sharing the same symbol a. Events of the form $B\langle a\rangle_k(\overline{v})$, referred to as *begin-events*, represent the beginning of an occurrence of the action with symbol $a \in \mathcal{A}$, uniquely identified by k, where each v_n (for any $n \in \{1, \ldots, |x|\}$) is the value assigned to the corresponding input variable x_n, with $v_n \in D_{x_n}$. Conversely, events of the form $E\langle a\rangle_k(\overline{w})$ are known as *end-events* and indicate the end of that occurrence, where each w_m (for any $m \in \{1, \ldots, |y|\}$) is the value of the output variable y_m, with $w_m \in D_{y_m}$.

Let τ be a finite sequence of finite sets of events, henceforth referred to as a *trace*, where each set of events corresponds to a discrete time point $t_j \in \mathbb{N}$, representing the j-th entry in the sequence. A trace emulates an execution of a system, representing the sequence of events that occur over time in a specific run:

$$\tau \;=\; [\{\varepsilon_1, \ldots, \varepsilon_p\}, \ldots, \{\varepsilon_q, \ldots, \varepsilon_r\}]$$

where $\varepsilon_j \in \mathcal{E}$ (for any $j \in \{1, \ldots, p, \ldots, q, \ldots, r\}$). We assume traces are well-formed: each $B\langle a\rangle_k(\ldots)$ has a unique matching $E\langle a\rangle_k(\ldots)$, with k appearing in exactly one $B\langle a\rangle$ and one $E\langle a\rangle$ event.

Definition (Action Occurrences)**.** The auxiliary *occurrences* operator returns the set of all occurrences of action a within trace τ, each characterized by its start and end time points, along with the associated input and output values. It is defined as follows:

$$\begin{aligned}\mathit{occurrences}(\tau, a) \;=\; \{(t_b, t_e, \overline{v}, \overline{w}) \mid & \textbf{ exists } k\,. \\ & \tau[t_b] \ni B\langle a\rangle_k(\overline{v}) \textbf{ and } \tau[t_e] \ni E\langle a\rangle_k(\overline{w})\}\end{aligned}$$

The *verification* of an ACTL formula φ with respect to a trace τ and a variable environment σ, denoted as $\tau, \sigma \models_{\text{ACTL}} \phi$, is defined inductively as follows:

(1) $\tau, \sigma \models_{\text{ACTL}} \neg\varphi \Leftrightarrow \textbf{not } \tau, \sigma \models_{\text{ACTL}} \varphi$
(2) $\tau, \sigma \models_{\text{ACTL}} \varphi_1 \vee \varphi_2 \Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi_1 \textbf{ or } \tau, \sigma \models_{\text{ACTL}} \varphi_2$
(3) $\tau, \sigma \models_{\text{ACTL}} \varphi_1 \wedge \varphi_2 \Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi_1 \textbf{ and } \tau, \sigma \models_{\text{ACTL}} \varphi_2$
(4) $\tau, \sigma \models_{\text{ACTL}} \varphi_1 \Rightarrow \varphi_2 \Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi_1 \textbf{ implies } \tau, \sigma \models_{\text{ACTL}} \varphi_2$
(5) $\tau, \sigma \models_{\text{ACTL}} \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y})\,.\,\varphi \Leftrightarrow$
$\quad \textbf{forall } (t_b, t_e, \overline{v}, \overline{w}) \in \mathit{occurrences}(\tau, a)\,.$
$\quad \tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{ACTL}} \varphi$
(6) $\tau, \sigma \models_{\text{ACTL}} \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y})\,.\,\varphi \Leftrightarrow$
$\quad \textbf{exists } (t_b, t_e, \overline{v}, \overline{w}) \in \mathit{occurrences}(\tau, a)\,.$
$\quad \tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{ACTL}} \varphi$
(7) $\tau, \sigma \models_{\text{ACTL}} x_1 = x_2 \Leftrightarrow \sigma(x_1) = \sigma(x_2)$
(8) $\tau, \sigma \models_{\text{ACTL}} r(i_1, i_2) \Leftrightarrow [\![r]\!](\sigma(i_1), \sigma(i_2))$

Informally, Item 5 states that $\forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi$ holds if, for every occurrence of action a in trace τ — spanning time points t_b to t_e with input values $\overline{v}$ and output values $\overline{w}$ — the formula φ holds when the environment σ is updated to map $\overline{x}$ to $\overline{v}$, $\overline{y}$ to $\overline{w}$, and i to the interval (t_b, t_e). Likewise, Item 6 states that $\exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi$ holds if there exists an occurrence of action a in trace τ such that, with the corresponding values and interval substituted into σ, the formula φ holds.

3.2 Specifying Distributed Hash Tables

This section demonstrates ACTL's expressive power by specifying operations and correctness properties of distributed protocols. Using actions with inputs, outputs, and time intervals, ACTL allows concise formalization of system behavior.

We formulate DHT operations as ACTL actions and specify value, key, and structural properties using ACTL formulas. Interval relations from ATL allow defining auxiliary predicates, such as $\mathit{In}(i_1, i_2)$ and $\mathit{Intersects}(i_1, i_2)$, to improve formula readability.

$$\mathit{In}(i_1, i_2) \iff \mathit{Starts}(i_1, i_2) \vee \mathit{During}(i_1, i_2) \vee \mathit{Finishes}(i_1, i_2)$$

$$\begin{aligned}\mathit{Intersects}(i_1, i_2) \iff{}& \mathit{Equals}(i_1, i_2) \vee \mathit{In}(i_1, i_2) \\ \vee{}& \mathit{In}(i_2, i_1) \vee \mathit{Overlaps}(i_1, i_2 \vee \mathit{Overlaps}(i_2, i_1)\end{aligned}$$

Some DHT properties hold only under specific conditions. For instance, two lookups for the same key should return the same value if the network is stable and no store operation occurs. We distinguish *runtime regimens*, indicating active operations, and *network states*, indicating the current system configuration.

In a network of uniquely identified nodes storing key–value pairs, we model functional, safety, and protocol-independent properties in ACTL. DHT operations are represented as actions parameterized by nodes n, keys k, and values v. Predicates can be modeled as actions with input only, while network states and regimens use only time intervals. The wildcard (–) marks parameters irrelevant to nested quantifiers or the property.

- **store**$(n, k, v) \overset{i}{\rightsquigarrow} (n')$: during interval i, the client requests node n to store the key-value pair (k, v) in the DHT; the operation completes at node n', which notifies the client upon completion.
- **lookup**$(n, k) \overset{i}{\rightsquigarrow} (n', v)$: during interval i, the client requests the value for key k from node n in the DHT; the operation completes at node n', which stores (k, v) and returns v to the client.
- **findnode**$(n, k) \overset{i}{\rightsquigarrow} (n', n'')$: during interval i, the client requests which node is responsible for key k from node n in the DHT; the operation completes at node n', which notifies the client that node n'' is responsible.
- **join**(n) ^{i}: during interval i, node n joins the network, resulting in a configuration in which node n is a member of the network.
- **leave**(n) ^{i}: during interval i, node n leaves the network, possibly triggering maintenance operations to update the network configuration and transfer stored mappings to other nodes; the operation results in a new configuration where n is no longer a member of the network.
- **responsible**(n, k) ^{i}: during interval i, node n is responsible for key k.
- **member**(n) ^{i}: during interval i, node n is a member of the network.
- **ideal**(n) ^{i}: during interval i, the network is in an *ideal* state (i.e. every node holds sufficient information to forward messages to every other node correctly).
- **readonly** ^{i}: during interval i, the network is in a *read only* regimen — neither membership status nor store operations occur (i.e., no store, remove, join, leave, or fail).
- **stable** ^{i}: during interval i, the network is in a *stable* regimen — no operations that alter the node membership status occur (i.e., no join, leave, or fail).

  When an interval is not in a *read only* or *stable* regimen, it is considered to be in a *standard* regimen. We do not define a predicate for this regimen, as it is not needed to model DHT properties.

(VALUE) **Lookup consistency**: if a lookup for key k reads a value v, then that value must have been previously assigned to that key by a store operation.

$$\begin{aligned}\forall\, \mathbf{lookup}(-, k) \overset{l}{\rightsquigarrow} (-, v) \,.\, \exists\, \mathbf{store}(-, k, v) \overset{s}{\rightsquigarrow} (-) \,.\, \\ \neg \mathit{Before}(l, s) \wedge \neg \mathit{Meets}(l, s)\end{aligned}$$

(VALUE) **Value consistency**: in an *ideal* state during a *read only* regimen, all lookup operations for a given key k return the same value v.

$$\begin{aligned}\forall\, \mathbf{lookup}(-, k) \overset{l_1}{\rightsquigarrow} (-, v_1) \,.\, \forall\, \mathbf{lookup}(-, k) \overset{l_2}{\rightsquigarrow} (-, v_2) \,.\, \\ \forall\, \mathbf{ideal}\ ^i \,.\, \forall\, \mathbf{readonly}\ ^r \,.\, \\ \Big(\big(\mathit{In}(l_1, i) \vee \mathit{Equals}(l_1, i)\big) \wedge \big(\mathit{In}(l_2, i) \vee \mathit{Equals}(l_2, i)\big) \\ \wedge \big(\mathit{In}(l_1, r) \vee \mathit{Equals}(l_1, r)\big) \wedge \big(\mathit{In}(l_2, r) \vee \mathit{Equals}(l_2, r)\big)\Big) \Rightarrow v_1 = v_2\end{aligned}$$

(VALUE) **Value freshness**: in an *ideal* state, all lookup operations for a key k return the value v written by the store operation that most recently terminated, one of its concurrent write operations, or an ongoing one.

$$\begin{aligned}\forall\, \mathbf{lookup}(-, k) \overset{l}{\rightsquigarrow} (-, v) \,.\, \exists\, \mathbf{ideal}\ ^i \,.\, \big(\mathit{In}(l, i) \vee \mathit{Equals}(l, i)\big) \\ \Rightarrow \exists\, \mathbf{store}(-, k, v) \overset{s_1}{\rightsquigarrow} (-) \,.\, \mathit{Intersects}(s_1, l) \\ \vee \big(\mathit{Before}(s_1, l) \wedge \forall\, \mathbf{store}(-, k, -) \overset{s_2}{\rightsquigarrow} (-) \,.\, \\ \big(s_1 \neq s_2 \wedge \mathit{Before}(s_2, l)\big) \Rightarrow \neg \mathit{Before}(s_1, s_2)\big)\end{aligned}$$

(KEY) **Key consistency**: in an *ideal* state during a *stable* regimen, all find node operations for a given key k agree on the same node n as being the responsible for the key.

$$\begin{aligned}\forall\, \mathbf{findnode}(-, k) \overset{f_1}{\rightsquigarrow} (-, n_1) \,.\, \forall\, \mathbf{findnode}(-, k) \overset{f_2}{\rightsquigarrow} (-, n_2) \,.\, \\ \forall\, \mathbf{ideal}\ ^i \,.\, \forall\, \mathbf{stable}\ ^s \,.\, \\ \Big(\big(\mathit{In}(f_1, i) \vee \mathit{Equals}(f_1, i)\big) \wedge \big(\mathit{In}(f_2, i) \vee \mathit{Equals}(f_2, i)\big) \\ \wedge \big(\mathit{In}(f_1, s) \vee \mathit{Equals}(f_1, s)\big) \wedge \big(\mathit{In}(f_2, s) \vee \mathit{Equals}(f_2, s)\big)\Big) \Rightarrow n_1 = n_2\end{aligned}$$

(KEY) **Findnode lookup consistency**: if a *find node* or *lookup* operation for key k returns node n or a value stored at that node, then

the node must have been responsible for that key either during the operation or at some earlier moment.

$$\forall\, \textbf{findnode}(-,k) \overset{f}{\rightsquigarrow} (-,n)\,.\,\exists\, \textbf{responsible}(n,k)^{\ r}\,.\, \mathit{Intersects}(f,r)$$
$$\wedge\ \forall\, \textbf{lookup}(-,k) \overset{l}{\rightsquigarrow} (n,-)\,.\,\exists\, \textbf{responsible}(n,k)^{\ r}\,.\, \mathit{Intersects}(l,r)$$

(KEY) **Responsibility transfer**: if any node n leaves the network, then that node cannot be responsible for any key unless it has joined the network in the meantime.

$$\forall\, \textbf{leave}(n)^{\ l}\,.\,\forall\, \textbf{findnode}(-,-) \overset{f}{\rightsquigarrow} (-,n)\,.\, \mathit{Before}(l,f)$$
$$\Rightarrow \exists\, \textbf{join}(n)^{\ j}\,.\, \mathit{Before}(l,j) \wedge \mathit{Before}(j,f)$$

(STRUCTURAL) **Membership guarantee**: the node n returned by any functional operation (e.g., store, lookup, find node) must have been a member for at least one moment, either before or during the execution of the operation.

$$\forall\, \textbf{store}(-,-,-) \overset{s}{\rightsquigarrow} (n)\,.\,\exists\, \textbf{member}(n)^{\ m}\,.\, \mathit{Intersects}(m,s)$$
$$\wedge\ \forall\, \textbf{lookup}(-,-) \overset{l}{\rightsquigarrow} (n,-)\,.\,\exists\, \textbf{member}(n)^{\ m}\,.\, \mathit{Intersects}(m,l)$$
$$\wedge\ \forall\, \textbf{findnode}(-,-) \overset{f}{\rightsquigarrow} (-,n)\,.\,\exists\, \textbf{member}(n)^{\ m}\,.\, \mathit{Intersects}(m,f)$$

(STRUCTURAL) **Reachability**: if any node n is a member during an *ideal* state, then all *find node* operations of the key k with the same identifier as the node must return the node.

$$\forall\, \textbf{findnode}(-,k) \overset{f}{\rightsquigarrow} (-,n)\,.\,\forall\, \textbf{ideal}^{\ i}\,.\,\forall\, \textbf{member}(k)^{\ m}\,.$$
$$\Big(\big(\mathit{In}(f,m) \vee \mathit{Equals}(f,m)\big) \wedge \big(\mathit{In}(m,i) \vee \mathit{Equals}(m,i)\big)\Big) \Rightarrow k = n$$

3.3 Decision Procedure

This section presents a decision procedure for verifying ACTL formulas over a given trace and variable environment. The procedure is formalized as a recursive algorithm that directly reflects the syntactic structure of ACTL, ensuring that each construct is evaluated in accordance with its formal semantics.

3.3.1 Pseudocode. The pseudocode in Algorithm 1 defines the function $\textsc{Check}(\tau, \sigma, \varphi)$, which returns *true* if and only if the formula φ is verified by the trace τ under the variable environment σ. The algorithm handles all syntactic forms of ACTL formulas.

To improve the efficiency of retrieving specific action occurrences from a trace, we introduce a map that associates each action with its set of occurrences within the trace. Formally, let $\mathcal{M}_\tau$ be an *action occurrences map*, i.e., a mapping from action symbols $a \in \mathcal{A}$ to the set of all values $(t_b, t_e, \overline{v}, \overline{w})$ corresponding to the occurrences of a, as determined by the begin and end event symbols $\varepsilon \in \mathcal{E}$ related to a in the trace τ. These tuples represent, respectively, the begin and end time points, and the associated input and output values of each occurrence:

$$\mathcal{M}_\tau : \mathcal{A} \to \wp(\mathbb{N} \times \mathbb{N} \times D^{|x|} \times D^{|y|})$$

where $\overline{x}$ and $\overline{y}$ would be the input and output parameters of action a, respectively. This construction of $\mathcal{M}$ is based on the same concept as the *occurrences* operator introduced in Section 3.1.3: for each action $a \in \mathcal{A}$, the set $\mathcal{M}_\tau[a]$ corresponds to $\mathit{occurrences}(\tau, a)$, serving as a practical representation grounded in the formalism previously introduced to support the semantics of ACTL. For simplicity, $\mathcal{M}_\tau[a]$ is omitted from the explicit arguments of $\textsc{Check}(\tau, \sigma, \varphi)$, though it is implicitly carried with τ through recursive calls.

Algorithm 1: The pseudocode of the trace verification algorithm for formulas of ACTL.

```
1  Function Check(τ, σ, φ):
2      switch φ do
3          case ¬φ′ do
4              return not Check(τ, σ, φ′)
5          case φ′ ∨ φ″ do
6              return Check(τ, σ, φ′) or Check(τ, σ, φ″)
7          case φ′ ∧ φ″ do
8              return Check(τ, σ, φ′) and Check(τ, σ, φ″)
9          case φ′ ⇒ φ″ do
10             return not Check(τ, σ, φ′) or Check(τ, σ, φ″)
11         case ∀a(x̄) ⇝^i (ȳ) . φ′ do
12             foreach (t_b, t_e, v̄, w̄) ∈ M_τ[a] do
13                 c ← Check(τ, σ[x̄ ↦ v̄, ȳ ↦ w̄, i ↦ (t_b, t_e)], φ′)
14                 if not c then return false
15             return true
16         case ∃a(x̄) ⇝^i (ȳ) . φ′ do
17             foreach (t_b, t_e, v̄, w̄) ∈ M_τ[a] do
18                 c ← Check(τ, σ[x̄ ↦ v̄, ȳ ↦ w̄, i ↦ (t_b, t_e)], φ′)
19                 if c then return true
20             return false
21         case x_1 = x_2 do
22             return σ(x_1) = σ(x_2)
23         case r(i_1, i_2) do
24             return [[r]](σ(i_1), σ(i_2))
```

The pseudocode in Algorithm 2 defines the $\textsc{PreprocessTrace}(\tau)$ function, which constructs the mapping $\mathcal{M}_\tau$ from the trace τ.

Algorithm 2: The pseudocode of the preprocessing algorithm for traces of ACTL.

```
1  Function PreprocessTrace(τ):
2      M_τ ← ∅; X ← ∅
3      for t ← 0 to |τ| − 1 do
4          foreach ε ∈ τ[t] do
5              switch ε do
6                  case B⟨a⟩_k(v̄) do
7                      X[k] ← (t, v̄)
8                  case E⟨a⟩_k(w̄) do
9                      (t_b, v̄) ← X[k]
10                     M_τ[a] ← M_τ[a] ∪ (t_b, t, v̄, w̄)
11     return M_τ
```

3.3.2 Complexity Analysis. We begin with a preprocessing step in which the trace τ is traversed to construct a mapping $\mathcal{M}_\tau$. Leveraging the presence of unique identifiers k, we can efficiently pair corresponding begin and end events during a single pass over the trace. To enable this single-pass construction, we additionally maintain an auxiliary mapping $\mathcal{X}$ that keeps track of *ongoing events*,

ensuring that matching identifiers k can be paired correctly. This results in a preprocessing time complexity of $O(e \cdot n)$, where $n = |\tau|$ is the number of time points in the trace, and e is the maximum number of events at any single time point. In the case where at most one event occurs per time point ($e = 1$), the complexity reduces to $O(n)$. Once the mapping $\mathcal{M}_\tau$ has been built, retrieving the set of occurrences corresponding to any action a requires only constant time, i.e., $O(1)$.

$$\textsc{PreprocessTrace}(\tau) \in O(e \cdot |\tau|)$$

Because action quantifiers are the constructs that are primarily responsible for the complexity of our trace verification algorithm, let q denote the total sum of nested action quantifiers in the formula being checked. In the worst-case scenario, when the formula is entirely composed of quantifiers, q is equivalent to the syntactic size of the formula, $q = |\varphi|$.

Each action quantifier in the formula retrieves the occurrences of the corresponding action in constant time, $O(1)$, using the precomputed mapping $\mathcal{M}_\tau$. For each occurrence found, the algorithm recursively checks a subformula. Therefore, the overall worst-case complexity is $O(o^q)$, where o is the maximum number of occurrences across all action quantifiers. In practice, o is typically small in real traces, and in fact is in the same order of magnitude as $|\tau|$.

$$\textsc{Check}(\tau, \sigma, \varphi) \in O(|\tau|^{|\varphi|})$$

The decision problem for ACTL is decidable, as the trace is finite. The complexity of the procedure is exponential in the number of nested quantifiers, and polynomial in the trace size for any fixed formula. Consequently, the verification problem lies in the class of EXPTIME-complete problems. In practice, formulas rarely consist solely of nested quantifiers, so the effective depth q is much smaller than the full syntactic size. We can assume that, when verifying properties, roughly only half of the constructs are quantifiers.

The main source of complexity arises from universal action quantifiers, since their subformula φ' must be checked for every occurrence of action a (up to o times), whereas existential quantifiers often stop early once a single satisfying occurrence is found. Conversely, when evaluating falsity — which is not the focus of this work — universal quantifiers may stop at the first counterexample, while existential ones may require checking all occurrences.

When checking property formulas, we can reasonably assume that only half of the quantifiers are universal, further reducing the practical complexity. Therefore, the worst-case complexity is rarely encountered in practice, making the effective complexity much lower than the theoretical upper bound.

Also, a simplification in action quantification — both reducing verbosity in property specifications (as seen in some cases in Section 3.2) and improving efficiency during formula checking— can be achieved by not distinguishing variables within quantifiers and deferring their equality constraints. This avoids exploring all occurrences of an action unnecessarily by pruning cases where a nested quantifier's variable is already bound to an incompatible value in the current environment:

$$\forall a(x_1) \overset{i_1}{\rightsquigarrow} (y_1) \,.\, \exists b(x_2) \overset{i_2}{\rightsquigarrow} (y_2) \,.\, x_1 = y_2 \;\equiv_{\text{opt}}$$
$$\forall a(x_1) \overset{i_1}{\rightsquigarrow} (y_1) \,.\, \exists b(x_2) \overset{i_2}{\rightsquigarrow} (x_1)$$

4 Implementation

4.1 Specification Language for ACTL

We developed an offline testing tool for ACTL that takes formulas specifying protocol properties. While human-readable formulas are concise, they are not machine-friendly. To enable parsing, we designed a formal specification language using only standard alphanumeric characters and a prefix notation with nested parentheses. This preserves readability and allows straightforward translation between mathematical formulas and the machine-readable format.

For example, the *lookup consistency* property for DHT protocols (Section 3.2) is shown in this language:

(VALUE) **Lookup consistency**: if a lookup for key k reads a value v, then that value must have been previously assigned to that key by a store operation.

$$\forall\, \textbf{lookup}(-, k) \overset{l}{\rightsquigarrow} (-, v) \,.\, \exists\, \textbf{store}(-, k, v) \overset{is}{\rightsquigarrow} (-) \,.\, \neg \mathit{Before}(l, s) \wedge \neg \mathit{Meets}(l, s)$$

```
(forall lookup l (- k) (- v)
    (exists store s (- k v) (-)
        (and
            (not (before l s))
            (not (meets l s))
)   )   )
```

Listing 1. Lookup consistency property in the ACTL specification language.

4.2 Instrumentation of a DHT implementation

We distinguish between *logs* — raw data collected from the distributed system — and *traces* — processed execution records ready for analysis. To obtain traces, these systems must be *instrumented* to record relevant runtime events. Instrumentation is challenging due to concurrency, message delays, clock skew, and failures. Fuzzers are often used to generate diverse execution scenarios. While developing instrumentation is beyond this work's scope, its quality directly affects analysis reliability.

We leverage pre-existing instrumented logs of OpenChord [9; 45]. Logs include timestamps, operation types, unique operation IDs, and relevant fields. They may also contain enriched data such as node membership intervals and inferred network states, which support both prior analyses and our study.

As defined in Section 3.1.3 and used in the ACTL decision procedure (Section 3.3), a trace τ is a sequence of event sets, which differs from the instrumented log format. To evaluate OpenChord logs, they must be transformed into this trace format by traversing the log and adapting its syntax.

4.3 ACTLChecker

ACTLChecker is the tool implementing the offline testing pipeline for distributed protocols using ACTL. Since it works on finite execution logs, verification is limited to safety properties. Figure 2 shows the system's general architecture. Users first specify protocol correctness properties in the machine-readable ACTL language. The protocol is then instrumented and executed to collect logs, with a fuzzer generating diverse scenarios to capture representative traces.

Logs may be preprocessed to include additional information or match ACTLChecker's input format. The tool then analyzes the traces against the specifications to check compliance. ACTLChecker is implemented in Python and openly available [1]. It is designed for testing complex, data-intensive systems like distributed protocols. Its implementation-independent modules require only properly formatted formulas and logs. The main pipeline modules are described below.

4.3.1 Formula Processor Module. The *Formula Processor* module takes as input a file describing a distributed protocol property specified in the formal language defined for the ACTL syntax in Section 4.1. It then parses this formula using the *Formula Parser* component and produces an Abstract Syntax Tree (AST) φ, represented as nested Python classes, to be used for the evaluation of the property. The leaves of this structure directly reflect the ACTL syntax defined in Section 3.1.2.

4.3.2 Log Processor Module. The *Log Processor* module takes as input an execution log generated during instrumentation, as described in Section 4.2. It parses this log using the *Log Parser* component to produce a trace τ, represented as a Python list of sets of events, consistent with the ACTL trace definition in Section 3.1.3. Each recorded operation is mapped to its corresponding begin and end events. Using the trace τ, the *Trace Preprocessor* component generates an auxiliary mapping $\mathcal{M}_\tau$, represented as a Python dictionary. This mapping provides efficient access to the occurrences of ACTL actions within the trace, as defined in Section 3.3.1, thereby reducing complexity during trace evaluation.

4.3.3 Monitor Module. The *Monitor* module is the core of the tool. It takes as input the AST φ derived from the property formula, the trace τ derived from the execution log, and the auxiliary mapping $\mathcal{M}_\tau$. Using its *Evaluator* component, the Monitor checks whether τ satisfies φ, recursively traversing the nested formula structure. Its efficient trace validation algorithm is the decision procedure for ACTL described in Section 3.3, which closely follows the semantics defined in Section 3.1.3. During evaluation, the environment σ is updated at each recursive call with the current variable bindings. The Monitor outputs a simple Boolean result: `true` if the trace satisfies the property, and `false` otherwise. Additionally, it supports several debugging options, such as limiting the number of log lines analyzed.

In practice, users invoke the Monitor by providing both the instrumented log and the property specification; the tool internally invokes the *Formula Processor* and *Log Processor* modules, abstracting away parsing and preprocessing details.

[1] https://github.com/jjasferreira/actl-checker

5 Evaluation

The evaluation of our implementation focuses on the same criteria that originally motivated its development: achieving efficient testing of distributed protocols, specifically through the monitoring of past-execution logs against ACTL-based specifications.

For assessing ACTLChecker in trace monitoring, we use DHT [37] as a case study. This choice is justified by two main factors:

- the availability of pre-collected logs from instrumented executions of an existing DHT implementation — OpenChord [45] — obtained in the work of Policarpo et al. [9], which eliminates the need for us to perform additional fuzzing or instrumentation.
- The existence of previously defined generic properties that DHT implementations are expected to satisfy, expressed in ACTL and presented as illustrative examples in Section 3.2 — naturally, these specifications are provided to the tool in the formal language defined in Section 4.1.

5.1 Functionality of ACTLChecker

We would like to determine whether the tool can be applied to validate concrete applications; that is, whether it effectively works in practice to verify or falsify properties against execution traces.

To test whether the tool functions correctly, we ran all the specifications of the properties described in Section 3.2, each against multiple traces of varying lengths that are *non-vacuous* with respect to the property; that is, traces containing both begin and end events corresponding to the actions relevant to the property. We consider a trace length to be the amount of events in a trace.

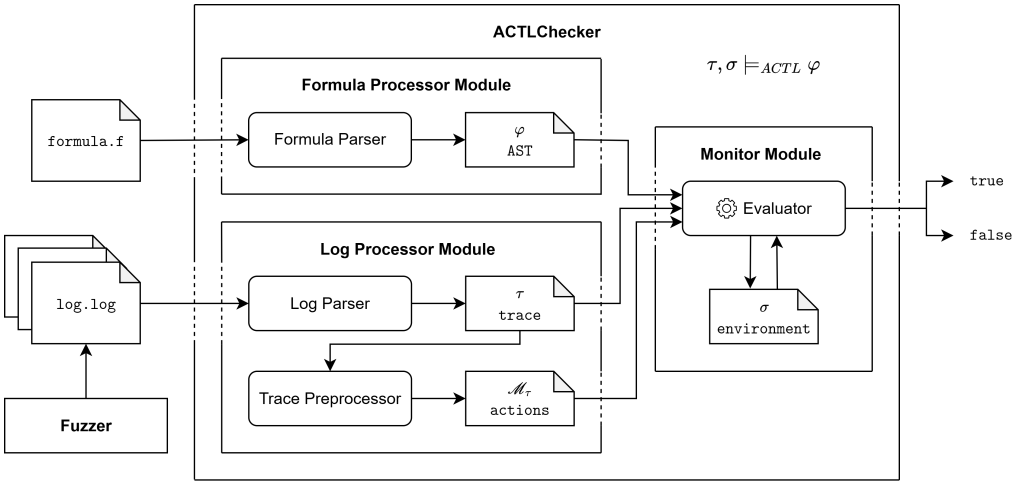
All of these properties are safety properties, since liveness properties cannot be verified over finite traces. Nevertheless, they cover different aspects of DHTs — keys, values, and structural characteristics. Table 1 below summarizes the results of this monitoring experiment across the eight properties:

Table 1. Properties, whether ACTLChecker can validate them against execution logs of OpenChord and the minimum vacuous trace lengths for each on a network of five nodes.

Property	ACTLChecker	Trace Length
Lookup consistency	`true`	150
Value consistency	`true`	vacuous
Value freshness	`true`	150
Key consistency	`true`	25
Findnode lookup consistency	`true`	25
Responsibility transfer	`false`	vacuous
Membership guarantee	`true`	25
Reachability	`true`	50

After running the tests, we observed that all properties passed except for *responsibility transfer*. This was expected and does not indicate an error or limitation of the implementation; rather, it reflects the behavior defined in Chord's theoretical algorithm [37].

Fig. 2. The general architecture of ACTLChecker.



These results are consistent with previous evaluations using Alloy for the same properties, as discussed in earlier work [9].

In summary, ACTLChecker can verify properties in traces, and it can detect property violations (e.g., for *responsibility transfer*), as expected.

6 Conclusions and Future Work

We address the challenges of expressive specification, representative observation, and efficient validation. In summary, the main contributions of this work are:

A. **ACTL: an action-based temporal logic.** We introduce a logic tailored for distributed protocols by combining interval-based reasoning with first-order quantification. Actions are modeled as first-class entities with inputs, outputs, and durations, enabling concise and expressive specifications.
B. **Formal specification of distributed protocols.** We demonstrate ACTL's expressiveness through correctness implementation -independent properties of DHTs.
C. **ACTLChecker: an efficient monitoring tool.** We implemented ACTLChecker in Python to verify execution traces against ACTL formulas. It parses specifications, encodes logs into traces, and efficiently checks property satisfaction using the proposed decision algorithm
D. **Evaluation on OpenChord traces.** Experiments on OpenChord logs confirm the tool's correctness and efficiency, validating all expected properties.

Further evaluation of ACTLChecker could reinforce the evidence of its scalability and practical applicability. Additional experiments with larger and more diverse logs — especially from networks with more nodes and keys — would help better characterize its performance limits. Such tests would also align more closely with real-world, large-scale industrial settings where execution logs tend to be substantially larger.

Beyond performance assessment, future work should also explore applying ACTLChecker to the testing and monitoring of other classes of distributed protocols, including concrete use cases such as alternative consensus mechanisms and blockchain protocols.

Beyond distributed protocols, the techniques presented in this work — namely the axiomatization of operations, the specification of correctness properties, and the monitoring of execution traces of implementations — can also be applied to other domains, including security and authentication protocols, communication protocols in IoT networks, blockchains, and protocols used in public administration systems.

Acknowledgments

This work was supported by the 'OptiGov' project, with ref. n. 2024.07385.IACDC (DOI: 10.54499/2024.07385.IACDC), fully funded by the 'Plano de Recuperação e Resiliência' (PRR) under the investment 'RE-C05-i08 - Ciência Mais Digital' (measure 'RE-C05-i08.m04'), framed within the financing agreement signed between the 'Estrutura de Missão Recuperar Portugal' (EMRP) and Fundação para a Ciência e a Tecnologia, I.P. (FCT) as an intermediary beneficiary.

I am deeply grateful to my supervisors Prof. José Fragoso Santos and Prof. Alessandro Gianola, to my family, colleagues, and friends for their unwavering support, guidance, and encouragement throughout this journey, without which this work would not have been possible.

References

[1] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems - concepts and designs (3. ed.)*, ser. International computer science series. Addison-Wesley-Longman, 2002. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/2252418288979304/Coulouris-Distributed_Systems.pdf
[2] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*, G. Gibson and N. Zeldovich, Eds. USENIX Association, 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
[3] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016. [Online]. Available: http://hdl.handle.net/10214/9769

[4] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining GHOST and casper," *CoRR*, vol. abs/2003.03052, 2020. [Online]. Available: https://arxiv.org/abs/2003.03052

[5] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J. Martin, "Revisiting fast practical byzantine fault tolerance," *CoRR*, vol. abs/1712.01367, 2017. [Online]. Available: http://arxiv.org/abs/1712.01367

[6] C. Jensen, H. Howard, and R. Mortier, "Examining raft's behaviour during partial network failures," in *HAOC '21: Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems, Virtual Event, United Kingdom, April 26, 2021*. ACM, 2021, pp. 11–17. [Online]. Available: https://doi.org/10.1145/3447851.3458739

[7] C. Cachin and M. Vukolic, "Blockchain consensus protocols in the wild," *CoRR*, vol. abs/1707.01873, 2017. [Online]. Available: http://arxiv.org/abs/1707.01873

[8] J. Neu, E. N. Tas, and D. Tse, "Ebb-and-flow protocols: A resolution of the availability-finality dilemma," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 446–465. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00045

[9] N. Policarpo, J. F. Santos, A. Cunha, J. Leitão, and P. Á. Costa, "Specifying distributed hash tables with allen temporal logic," in *13th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*. IEEE, 2025, pp. 63–73. [Online]. Available: https://doi.org/10.1109/FormaliSE66629.2025.00013

[10] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, 1983. [Online]. Available: https://doi.org/10.1145/182.358434

[11] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. [Online]. Available: https://doi.org/10.1109/SFCS.1977.32

[12] I. M. Hodkinson, F. Wolter, and M. Zakharyaschev, "Decidable fragment of first-order temporal logics," *Ann. Pure Appl. Log.*, vol. 106, no. 1-3, pp. 85–134, 2000. [Online]. Available: https://doi.org/10.1016/S0168-0072(00)00018-X

[13] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, 1994. [Online]. Available: https://doi.org/10.1145/177492.177726

[14] M. Shanahan, "The event calculus explained," in *Artificial Intelligence Today: Recent Trends and Developments*, ser. Lecture Notes in Computer Science, M. J. Wooldridge and M. M. Veloso, Eds. Springer, 1999, vol. 1600, pp. 409–430. [Online]. Available: https://doi.org/10.1007/3-540-48317-9_17

[15] J. F. Allen, "Towards a general theory of action and time," in *The Language of Time - A Reader*, I. Mani, J. Pustejovsky, and R. J. Gaizauskas, Eds. Oxford University Press, 2005, pp. 251–276. [Online]. Available: https://doi.org/10.1016/0004-3702(84)90008-0

[16] ——, "An interval-based representation of temporal knowledge," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence, IJCAI '81, Vancouver, BC, Canada, August 24-28, 1981*, P. J. Hayes, Ed. William Kaufmann, 1981, pp. 221–226. [Online]. Available: http://ijcai.org/Proceedings/81-1/Papers/045.pdf

[17] M. B. Vilain and H. A. Kautz, "Constraint propagation algorithms for temporal reasoning," in *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, USA, August 11-15, 1986. Volume 1: Science*, T. Kehler, Ed. Morgan Kaufmann, 1986, pp. 377–382. [Online]. Available: http://www.aaai.org/Library/AAAI/1986/aaai86-063.php

[18] J. Pustejovsky, J. M. Castaño, R. Ingria, R. Saurí, R. J. Gaizauskas, A. Setzer, G. Katz, and D. R. Radev, "Timeml: Robust specification of event and temporal expressions in text," in *New Directions in Question Answering, Papers from 2003 AAAI Spring Symposium, Stanford University, Stanford, CA, USA*, M. T. Maybury, Ed. AAAI Press, 2003, pp. 28–34.

[19] J. Condotta, G. Ligozat, and M. Saade, "Qualitative constraints for job shop scheduling," in *Benchmarking of Qualitative Spatial and Temporal Reasoning Systems, Papers from the 2009 AAAI Spring Symposium, Technical Report SS-09-02, Stanford, California, USA, March 23-25, 2009*. AAAI, 2009, pp. 1–4. [Online]. Available: http://www.aaai.org/Library/Symposia/Spring/2009/ss09-02-001.php

[20] G. Zhang, X. Li, Y. Huang, and L. Cui, "Temporal cohort logic," in *AMIA 2022, American Medical Informatics Association Annual Symposium, Washington, DC, USA, November 5-9, 2022*. AMIA, 2022. [Online]. Available: https://knowledge.amia.org/76677-amia-1.4637602/f006-1.4642154/f006-1.4642155/645-1.4642165/1009-1.4642162

[21] A. Hakeem, Y. Sheikh, and M. Shah, "CASEE: A hierarchical event representation for the analysis of videos," in *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, D. L. McGuinness and G. Ferguson, Eds. AAAI Press / The MIT Press, 2004, pp. 263–268. [Online]. Available: http://www.aaai.org/Library/AAAI/2004/aaai04-042.php

[22] J. Renz and B. Nebel, "Qualitative spatial reasoning using constraint calculi," in *Handbook of Spatial Logics*, M. Aiello, I. Pratt-Hartmann, and J. van Benthem, Eds. Springer, 2007, pp. 161–215. [Online]. Available: https://doi.org/10.1007/978-1-4020-5587-4_4

[23] G. Rosu and S. Bensalem, "Allen linear (interval) temporal logic - translation to LTL and monitor synthesis," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 263–277. [Online]. Available: https://doi.org/10.1007/11817963_25

[24] T. Janhunen and M. Sioutis, "Allen's interval algebra makes the difference," in *Declarative Programming and Knowledge Management - Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9-12, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, P. Hofstedt, S. Abreu, U. John, H. Kuchen, and D. Seipel, Eds., vol. 12057. Springer, 2019, pp. 89–98. [Online]. Available: https://doi.org/10.1007/978-3-030-46714-2_6

[25] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification (preliminary report)," in *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*. IEEE Computer Society, 1986, pp. 332–344.

[26] E. M. Clarke, O. Grumberg, D. Kroening, D. A. Peled, and H. Veith, *Model checking, 2nd Edition*. MIT Press, 2018. [Online]. Available: https://mitpress.mit.edu/books/model-checking-second-edition

[27] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, 1991. [Online]. Available: https://doi.org/10.1016/0304-3975(91)90224-P

[28] D. A. Basin, F. Klaedtke, and E. Zalinescu, "The monpoly monitoring tool," in *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, ser. Kalpa Publications in Computing, G. Reger and K. Havelund, Eds., vol. 3. EasyChair, 2017, pp. 19–28. [Online]. Available: https://doi.org/10.29007/89hs

[29] F. Chen and G. Rosu, "Mop: an efficient and generic runtime verification framework," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds. ACM, 2007, pp. 569–588. [Online]. Available: https://doi.org/10.1145/1297027.1297069

[30] A. Gianola, M. Montali, and S. Winkler, "Linear-time verification of data-aware processes modulo theories via covers and automata," in *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, M. J. Wooldridge, J. G. Dy, and S. Natarajan, Eds. AAAI Press, 2024, pp. 10 525–10 534. [Online]. Available: https://doi.org/10.1609/aaai.v38i9.28922

[31] D. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev, "Fragments of first-order temporal logics," in *Many-Dimensional Modal Logics: Theory and Applications*, ser. Studies in Logic and the Foundations of Mathematics. Elsevier, 2003, vol. 148, pp. 465–545.

[32] L. Geatti, A. Gianola, and N. Gigante, "Linear temporal logic modulo theories over finite traces," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, L. D. Raedt, Ed. ijcai.org, 2022, pp. 2641–2647. [Online]. Available: https://doi.org/10.24963/ijcai.2022/366

[33] D. Jackson, "Alloy: a language and tool for exploring software designs," *Commun. ACM*, vol. 62, no. 9, pp. 66–76, 2019. [Online]. Available: https://doi.org/10.1145/3338843

[34] L. Lamport, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. [Online]. Available: http://research.microsoft.com/users/lamport/tla/book.html

[35] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services uses formal methods," *Communications of the ACM*, vol. 58, pp. 66–73, 2015.

[36] H. Cirstea, M. A. Kuppe, B. Loillier, and S. Merz, "Validating traces of distributed programs against tla$^+$ specifications," in *Software Engineering and Formal Methods - 22nd International Conference, SEFM 2024, Aveiro, Portugal, November 6-8, 2024, Proceedings*, ser. Lecture Notes in Computer Science, A. Madeira and A. Knapp, Eds., vol. 15280. Springer, 2024, pp. 126–143. [Online]. Available: https://doi.org/10.1007/978-3-031-77382-2_8

[37] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*, R. L. Cruz and G. Varghese, Eds. ACM, 2001, pp. 149–160. [Online]. Available: https://doi.org/10.1145/383059.383071

[38] P. Zave, "Using lightweight modeling to understand chord," *Comput. Commun. Rev.*, vol. 42, no. 2, pp. 49–57, 2012. [Online]. Available: https://doi.org/10.1145/

2185376.2185383
[39] Y. Howard, S. Gruner, A. M. Gravell, C. Ferreira, and J. C. Augusto, "Model-based trace-checking," *CoRR*, vol. abs/1111.2825, 2011. [Online]. Available: http://arxiv.org/abs/1111.2825
[40] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.
[41] P. Zave, "A practical comparison of alloy and spin," *Formal Aspects Comput.*, vol. 27, no. 2, pp. 239–253, 2015. [Online]. Available: https://doi.org/10.1007/s00165-014-0302-2
[42] T. Lu, S. Merz, and C. Weidenbach, "Towards verification of the pastry protocol using tla $^+$," in *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings*, ser. Lecture Notes in Computer Science, R. Bruni and J. Dingel, Eds., vol. 6722. Springer, 2011, pp. 244–258. [Online]. Available: https://doi.org/10.1007/978-3-642-21461-5_16
[43] N. Azmy, S. Merz, and C. Weidenbach, "A machine-checked correctness proof for pastry," *Sci. Comput. Program.*, vol. 158, pp. 64–80, 2018. [Online]. Available: https://doi.org/10.1016/j.scico.2017.08.003
[44] P. Zave, "Reasoning about identifier spaces: How to make chord correct," *IEEE Trans. Software Eng.*, vol. 43, no. 12, pp. 1144–1156, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2017.2655056
[45] S. Kaffille and K. Loesing, "OpenChord: Implementation of the Chord DHT," https://open-chord.sourceforge.net/, 2008. [Online]. Available: https://open-chord.sourceforge.net/