TÉCNICO LISBOA

Specifying and Testing Distributed Protocols with Action Temporal Logic

José João Alves dos Santos Ferreira

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Alessandro Gianola
Prof. José Faustino Fragoso Femenin dos Santos

Examination Committee

Chairperson: Prof. Miguel Ângelo Marques de Matos
Supervisor: Prof. Alessandro Gianola
Member of the Committee: Prof. António José dos Reis Morgado

October 2025

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

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 sincerely thankful to my dissertation supervisors, Prof. José Fragoso Santos and Prof. Alessandro Gianola, for their invaluable guidance, insight, and knowledge, as well as for their patience and dedication. Your mentorship has been fundamental to the completion of this work and to my growth as a researcher.

My heartfelt thanks also go to Nuno Policarpo, whose earlier work inspired this project. His contributions to the development of the tool presented here and his continued help were instrumental in shaping the outcome of this research.

I would like to express my deepest gratitude to my parents and my brother for their unwavering love, friendship, encouragement, and support throughout all these years. Your constant presence through every challenge and triumph has been my greatest strength. Without you, this project would not have been possible. I am also deeply grateful to my grandparents, aunts, uncles, and cousins for their understanding, kindness, and continued support along the way.

Finally, I would like to thank all my friends and colleagues who have been part of this journey. Your companionship, support, and countless moments of laughter made even the most difficult times easier. A special thanks to João, Francisco, and Daniel for their friendship and encouragement through both the highs and lows.

To each and every one of you, thank you.

Abstract

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 2PC and DHT. 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.

Keywords

Offline Monitoring; Distributed Protocols; Two-Phase Commit; Distributed Hash Tables; Allen Temporal Logic; First-Order Linear Temporal Logic

Resumo

Os sistemas distribuídos constituem a espinha dorsal de infraestruturas modernas, como serviços financeiros, plataformas na cloud e redes blockchain. Garantir a correção dos seus protocolos de coordenação e tolerância a falhas é um desafio, pois implementações reais frequentemente divergem de designs formalmente verificados, conduzindo a falhas subtis mas críticas. A monitorização offline permite validar implementações de protocolos analisando registos de execução face a especificações formais. Ferramentas existentes, como TLA+ e Alloy, enfrentam problemas de escalabilidade, e nenhuma linguagem formal atual captura de forma eficiente as dependências temporais e de dados em protocolos distribuídos para monitorização. Este trabalho aborda estes desafios com a introdução de um quadro para testar protocolos distribuídos, que inclui uma lógica temporal adaptada para especificação, Action Temporal Logic (ACTL), integrando o raciocínio baseado em intervalos de ATL com a quantificação e expressividade de FOLTL. ACTL fornece uma sintaxe intuitiva centrada na acção para especificar relações temporais, causais e dependentes de dados entre operações do protocolo, demonstrada com exemplos de 2PC e DHT. O quadro inclui também o ACTLChecker, uma ferramenta de monitorização que interpreta especificações formais e registos de execução para verificar se um traço satisfaz uma fórmula ACTL, usando um algoritmo de decisão com pré-processamento otimizado. Avaliamos a abordagem com registos do OpenChord, uma implementação industrial do protocolo Chord, uma DHT. O ACTLChecker deteta violações de protocolo e verifica de forma eficiente registos conformes, melhorando tempo de verificação e escalabilidade em relação a abordagens baseadas em Alloy.

Palavras Chave

Monitorização Offline; Protocolos Distribuídos; Commit em Duas Fases; Tabelas de Dispersão Distribuídas; Lógica Temporal de Allen; Lógica Temporal Linear de Primeira Ordem

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

2PC	Two-Phase Commit
3PC	Three-Phase Commit
ACTL	Action Temporal Logic
AI	Artificial Intelligence
API	Application Programming Interface
ASP	Answer Set Programming
AST	Abstract Syntax Tree
ATL	Allen Temporal Logic
CSP	Constraint Satisfaction Problem
dFOATL	Discrete First-Order Allen Temporal Logic
DHT	Distributed Hash Table
DSL	Domain Specific Language
EC	Event Calculus
FOL	First-Order Logic
FOLTL	First-Order Linear Temporal Logic
LTL	Linear Temporal Logic
NLP	Natural Language Processing
PL	Propositional Logic
SAT	Boolean Satisfiability
SMT	Satisfiability Modulo Theories
TLA	Temporal Logic of Actions

1

Introduction

Contents

1.1 Motivation

In today's world, software systems are increasingly reliant on distributed operations. These distributed systems form the foundation of many essential services, such as large-scale social networks, financial infrastructures, cloud computing and storage platforms, and blockchain networks. Since these systems support critical applications, they must operate reliably and remain available at all times.

Distributed systems rely on coordination protocols to ensure correct communication and consistency among nodes. These distributed protocols — such as Two-Phase Commit (2PC), Distributed Hash

Tables (DHTs), and consensus mechanisms [1] — are fundamental for maintaining reliability and fault tolerance.

Ensuring that distributed protocols behave correctly is a core challenge. Ideally, such protocols should be formally verified to guarantee that they meet their specifications under all conditions. However, even when algorithms are proved correct in theory, their real-world implementations may deviate from the intended behavior, exposing critical flaws that compromise the correctness guarantees of the underlying algorithms. Verifying production-level code with existing formal tools remains time-consuming and technically demanding, which often restricts formal proofs to the abstract algorithm rather than the real implementation. As a result, correctness is usually guaranteed only at the design level, leaving room for implementation errors.

Implementations of widely known consensus algorithms, such as FaB Paxos [2] and Raft [3], have been put to the test and found to contain severe flaws [4, 5]. More recently, Tendermint [6], used in the Cosmos blockchain, and Gasper [7], employed by Ethereum, are examples of proof-of-stake consensus protocols whose implementations were also discovered to exhibit critical errors [8, 9].

There is a rich landscape of various techniques for validating distributed protocols, each with distinct trade-offs:

(a) *deductive verification* using proof assistants such as Coq [10, 11], and TLAPS [12, 13], which offers strong mathematical guarantees but is complex and time-consuming for large systems.

(b) *model checking* [14, 15], which explores all possible system executions to detect property violations, but where scalability is limited by state-space explosion.

(c) *fuzzing* [16, 17], which generates random workloads to expose unexpected behaviors, but it provides no completeness guarantees; and

(d) *monitoring* [18, 19], which analyzes execution logs to check whether system behavior complies with formal specifications, being practical for deployed systems but detecting issues only after execution.

This work focuses on *offline monitoring*, where traces collected from system executions are analyzed once the system has finished running. This approach is generally more tractable than online monitoring, as it operates on finite and complete traces, and eliminates the need to reason under the uncertainty introduced by ongoing or incomplete executions, thereby enabling more comprehensive and deterministic evaluation of system behaviors.

Monitoring distributed protocols poses several difficulties. Expressing their properties requires reasoning about the timing, ordering, and dependencies between operations, including data relationships and causal links. Obtaining reliable logs is also challenging: events must be correlated and ordered

across multiple nodes despite delays, clock offsets, and possible failures. While constructing such logs is beyond the scope of this work, it is essential for the accuracy of any trace-based analysis.

Existing offline monitoring tools, such as TLA+ [20] and Alloy [21], often suffer from scalability and efficiency limitations when handling large execution traces. In particular, Alloy requires dynamic or sequential behaviors to be encoded through ad-hoc idioms, making protocol specifications cumbersome, and it was not designed to scale for large scopes. Our goal is to achieve greater efficiency and scalability than the work of Policarpo et al. [22], which employs an Alloy-based monitor to validate an implementation of the Chord [23] distributed protocol.

Overall, this work is motivated by three main challenges:

1. **Specification:** how to express protocol properties in an intuitive yet expressive way.
2. **Observation:** how to obtain and interpret representative execution logs of protocols.
3. **Validation:** how to efficiently verify traces against formal specifications at scale.

1.2 Goals

To bridge the gap between the complexity of distributed systems and the limitations of existing monitoring and testing approaches, we propose a new framework for testing distributed protocols. Our goals are directly aligned with the challenges identified above, each targeting a specific aspect of the specification, observation, and validation process.

1. **Design a new logic for specifying properties of distributed protocols in an intuitive way, balancing formal rigor with expressiveness.**

   When assessing which formalism framework to choose for writing these formulas, we examine several existing logics for reasoning about time-spanning operations and their interrelations, such as Allen Temporal Logic (ATL) [24], Linear Temporal Logic (LTL) [25], First-Order Linear Temporal Logic (FOLTL) [26], Temporal Logic of Actions (TLA) [20], and Event Calculus (EC) [27].

   These approaches are either too abstract or too rigid for specifying and monitoring real-world distributed protocols. We focus on ATL, which is expressive for reasoning about temporal relations between intervals but lacks essential first-order constructs, and FOLTL, which is powerful for data-rich temporal reasoning with quantification but syntactically heavy and less intuitive for modeling protocol-level event relations.

   These logics alone are insufficient. We therefore aim to design a more suitable and intuitive framework for distributed protocols — a new Domain Specific Language (DSL) that balances formal rigor with practical expressiveness.

2. **Instrument the system to collect logs and create a trace format that best supports our framework.**

   Instrumentation refers to the process of adding mechanisms to a system to record detailed information about its execution. This allows for later analysis and verification of system behavior.

   In our case, we use logs previously obtained from executions of OpenChord [28], an industrial implementation of the distributed Chord protocol [23], a DHT, to collect detailed records of protocol activity. These logs have been enriched with additional information required for the formalization of DHT properties in earlier work [22]. We then transform these logs into a format that aligns with our monitoring needs and the newly designed DSL.

3. **Develop an efficient monitoring tool that, given a log and a formula, determines whether the trace satisfies the property.**

   This involves defining an efficient decision procedure, especially for exploring variable quantification, and preprocessing traces for efficiency purposes. The goal is to make the tool practical and scalable.

   The goals here are to define a formal specification language for property formulas aligned with the syntax of the DSL, define a notation for traces that aligns with our trace definition and supports the decision procedure, and lastly build a monitoring tool, including: an Abstract Syntax Tree (AST), whose nodes follow the DSL syntax, a *Formula Processor* to parse property formulas into an AST, a *Log Processor* to parse execution logs into traces, and A *Monitor* that integrates both modules to validate traces against formulas.

1.3 Contributions

The concrete contributions of this work are:

A. **Action Temporal Logic (ACTL): definition of a new fragment based on actions.**

   We conduct an iterative design process to develop the most suitable DSL, combining the strengths of ATL and FOLTL. Through this process, we introduce Discrete First-Order Allen Temporal Logic (dFOATL), a hybrid logic that integrates Allen's interval relations and their relative temporal reasoning capabilities which are useful for specifying protocol properties, with the precise, discrete point–based reasoning, quantification, and other expressive constructs of first-order logic. Building upon dFOATL, we ultimately derive ACTL, our final and refined specification language.

   ACTL introduces actions as a direct formalism for operations that take inputs, produce outputs, and occur over intervals between discrete time points. We further refine and study this logic, as

it provides the most suitable syntax for specifying distributed protocol properties and can express temporal, causal, and data-related dependencies in a unified way.

The definition of ACTL includes its full formal syntactic and semantic specification, an efficient decision algorithm for verifying properties against execution traces, formal proofs, and a formal encoding relating it to dFOATL, the logic from which it derives as a variant.

B. **Specification of properties of distributed protocols using temporal logics.**

We specify properties of 2PC, a coordination protocol ensuring atomic commitment, and of DHT, a core component of many peer-to-peer and blockchain systems, using logics for temporal reasoning, particularly the newly designed ACTL, but also FOLTL.

These case studies are used throughout the document to demonstrate the application of temporal logics as a framework for formalizing correctness properties of distributed protocols.

C. **ACTLChecker: implementation of an efficient monitoring tool for ACTL formulas.**

Implemented in Python, this tool is particularly suited for testing executions of complex, data-intensive systems with interdependent operations and numerous properties to verify, as is the case with distributed protocols. It consists mainly of three modules:

A. the *Formula Processor*, which parses a formula written in the formal specification language defined for ACTL properties and produces an AST, whose classes closely follow the invented ACTL syntax;

B. the *Log Processor*, which parses execution logs obtained through instrumentation and produces a trace in accordance with our ACTL trace definition, mapping recorded operations into corresponding events, as well as auxiliary mappings that reduce complexity during property verification; and

C. the *Monitor*, the main module, which, given an instrumented log and a formula written in the specification language, calls both processors to parse the formula and the log, producing an AST and a trace, respectively. It then validates the formula on the trace using a newly developed and efficient decision procedure, outputting whether the trace satisfies the property.

D. **Evaluation: monitoring of DHT properties against traces using ACTLChecker.**

We evaluate our approach using execution traces of OpenChord, as a case study, and conduct a comprehensive evaluation of monitoring DHT property formulas against these traces, studying how well the tool performs validation and scales with increasing trace sizes. We also assess the tool's relevance and applicability for future use.

Since large log sizes are often crucial to ensuring that all types of operations are captured and that the traces are representative of the distributed system's behavior, and because our approach

scales more effectively with increasing trace sizes, we provide stronger guarantees for the potential monitoring of such protocols compared to previous Alloy-based work.

An important aspect of this research is the *scalability* of the trace validation algorithm. Distributed systems typically generate very large traces, and using specification languages with higher degrees of freedom would make validation prohibitively expensive. ACTL and its corresponding formal specification language represent a *sweet spot*: they feature an action-focused syntax that enables the specification of complex system properties while maintaining an internal structure that allows efficient validation. We therefore explore a trade-off between expressiveness and efficiency that, to our knowledge, has not been previously addressed in the literature.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the essential background on the temporal logics that inspired the development of our own formalism, namely ATL and FOLTL, as well as on two representative distributed protocols — 2PC and DHT — along with their correctness properties. Chapter 3 surveys related work on time-reasoning logics, distributed protocol specification, and DHT testing. Chapter 4 develops the formal syntax and semantics of dFOATL, a discrete first-order variant of ATL, as well as illustrative examples and a decision procedure, which serve as an intermediate step toward defining a more practical formalism for protocol specification and testing. Chapter 5 introduces ACTL, detailing its syntax and semantics, presenting specifications of protocols, and describing the translation from ACTL to dFOATL, as well as the decision procedure, pseudocode, and complexity analysis. Chapter 6 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. Chapter 7 presents the experimental evaluation of our work, assessing ACTLChecker's functionality. Finally, Chapter 8 concludes the thesis by summarizing the main contributions and outlining directions for future research, followed by the references.

1.5 Publications

Some of the preliminary work underlying this thesis has already been published in the short paper listed below. That work focuses on specifying properties of the distributed protocols 2PC and DHT in 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

Background

Contents

This chapter provides essential background on relevant temporal logic fragments that have been used in the fields of formal verification and Artificial Intelligence (AI), and have influenced the subsequent development of our own temporal logic fragment for specifying and verifying distributed protocols, namely Allen Temporal Logic (ATL) in Section 2.1 and First-Order Linear Temporal Logic (FOLTL), as well as its predecessors, in Section 2.2.

It then introduces two distributed protocols — Two-Phase Commit (2PC) in Section 2.3 and Distributed Hash Table (DHT) in Section 2.4 — along with fundamental notions, operations, and illustrative correctness properties to be verified. These properties fall into two categories: (a) safety, asserting that nothing bad ever happens; and (b) liveness, ensuring that something good eventually happens.

The properties are first specified in FOLTL and later reformulated using the latest temporal logic

fragment developed through our iterative work (Section 5.3). This fragment serves as the basis for our evaluation of property verification for the DHT protocol in Chapter 7.

2.1 Allen Temporal Logic

Allen Temporal Logic (ATL) [24, 29], sometimes called Allen's Interval Algebra, is an interval-based temporal logic originally introduced by Allen [30] to capture the temporal reasoning required in AI applications. It provides a rigorous framework for representing incomplete and relative temporal information, for instance, by expressing that an interval i_1 occurs before or during another interval i_2. Due to its expressive power and suitability for modeling time-dependent phenomena, ATL is an established formalism for temporal reasoning.

Its core components are temporal intervals i and propositions p. Intervals are continuous, occur only once, and can be infinitely subdivided into sub-intervals. In contrast, propositions may hold over various non-continuous time periods, distinguishing them from intervals. ATL models the uncertainty and incompleteness inherent in time-spanning events by specifying relations between intervals rather than precise time points, where $i = [t_1, t_2]$.

Time is interpreted as a continuum represented by intervals. Importantly, ATL is agnostic to whether time is discrete or continuous: it is defined over a linearly ordered domain and does not depend on time granularity. The underlying timeline can be any linearly ordered set — if interpreted over the integers, one obtains a discrete version ($i = [t_1, t_2] \subseteq \mathbb{N}$), and if interpreted over the reals, a dense or continuous version ($i = [t_1, t_2] \subseteq \mathbb{R}$). Allen's definitions and reasoning patterns hold for any such linear order.

The relations between time intervals are qualitative and expressed through a set of seven fundamental binary interval predicates: $Before(i_1, i_2)$, $Meets(i_1, i_2)$, $Overlaps(i_1, i_2)$, $Starts(i_1, i_2)$, $During(i_1, i_2)$, $Finishes(i_1, i_2)$, and $Equals(i_1, i_2)$. From these, inverse and more complex relations can be derived. In addition, ATL defines predicates that associate propositions with intervals, such as $Holds(p, i)$ and $Occurs(p, i)$.

An ATL formula is a finite Boolean combination of such relations. Some variants of ATL include the inverse interval relations $After(i_1, i_2)$, $MetBy(i_1, i_2)$, $OverlappedBy(i_1, i_2)$, as well as $StartedBy(i_1, i_2)$, $Contains(i_1, i_2)$, and $FinishedBy(i_1, i_2)$. However, these do not increase expressiveness, as they merely swap the arguments of the already existing interval predicates.

Table 2.1 provides a summary of the ATL predicates, which are illustrated in Figure 2.1 and Figure 2.2. The formal syntax and semantics of ATL were defined in earlier work [31].

Inspired by the concepts of ATL, our own logic fragment for testing distributed protocols, designed later on through iterative work, adopts the notion of intervals along with interval and proposition relations for the specification of properties using formulas. We too represent these relations using predicates,

Table 2.1: Summary of the semantics of ATL predicates.

Predicate	Meaning
$Before(i_1, i_2)$	Interval i_1 ends before i_2 starts
$Meets(i_1, i_2)$	Interval i_2 starts immediately after i_1 ends
$Overlaps(i_1, i_2)$	Interval i_1 starts before i_2 and ends during i_2
$Starts(i_1, i_2)$	Interval i_1 starts simultaneously with i_2 and ends earlier than i_2
$During(i_1, i_2)$	Interval i_1 is strictly contained within i_2
$Finishes(i_1, i_2)$	Interval i_1 ends simultaneously with i_2 and starts later than i_2
$Equals(i_1, i_2)$	Interval i_1 and i_2 coincide
$Holds(p, i)$	Proposition p holds throughout interval i
$Occurs(p, i)$	Proposition p holds at least once during interval i

Figure 2.1: Illustration of the interval predicates of ATL relations.

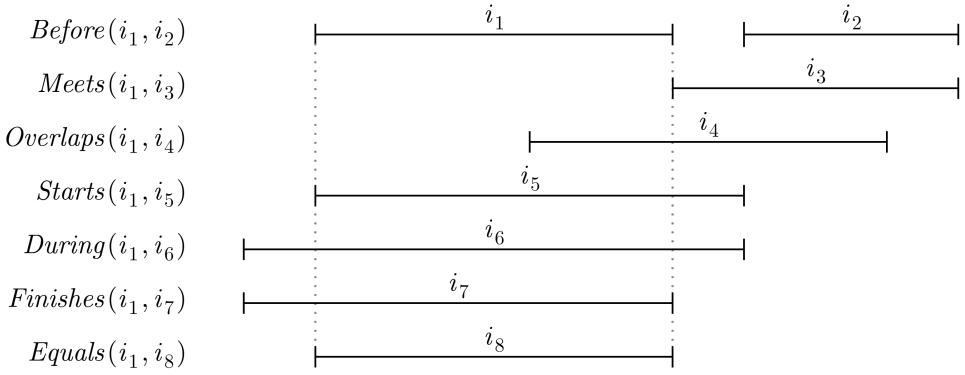
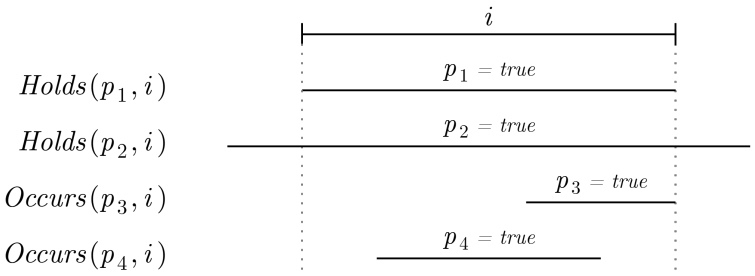


Figure 2.2: Illustration of the proposition predicates of ATL relations.



enabling effective reasoning about the relative ordering of operations in protocol traces.

2.2 First-Order Linear Temporal Logic

First-Order Linear Temporal Logic (FOLTL) represents an extension of earlier logical systems, combining the strengths of classical logic, temporal reasoning, and first-order constructs to address the dynamic nature of complex systems that evolve over time. To understand it, it is essential to consider its roots in

these previously defined logical systems.

Propositional Logic (PL) is a branch of logic that deals with propositions — statements represented in a formal language using symbols (e.g., p, q) that are either *true* or *false* — as well as logical operators ($\wedge$ for *conjunction*, $\rightarrow$ for *implication*, $\leftrightarrow$ for *equivalence*, $\top$ for *true*, $\bot$ for *false*, among others), derived from the basic logical connectives $\neg$ for *negation* and $\vee$ for *disjunction*. Combined, they form compound propositions and provide a framework for reasoning about the truth or falsity of statements, serving as the foundation for more expressive logics.

First-Order Logic (FOL) extends PL, incorporating all its logical operators, by introducing additional expressive constructs that enable reasoning about objects and their relationships within a domain, including predicates (e.g., $p(t_1, t_2)$), which represent properties or relations among terms; terms (e.g., x, y), which may be variables, constants, or the result of applying functions to other terms; and quantifiers ($\forall$ for *universal quantification* and $\exists$ for *existential quantification*). Together, these constructs allow FOL to formalize and reason about complex properties and relationships that cannot be captured in PL.

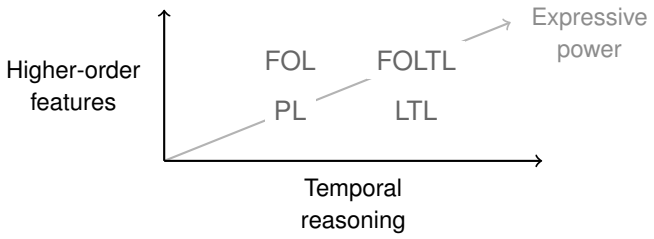
First-Order Logic (FOL) extends PL, incorporating all its logical operators, by introducing predicates (e.g., $p(t_1, t_2)$, $q(u_1, u_2, u_3)$), terms, which can be variables (e.g., x, y), constants, functions, and quantifiers ($\forall$ for *universal quantification* and $\exists$ for *existential quantification*), allowing for the expression of statements about objects within a domain. FOL provides a framework for reasoning about properties and relationships, enabling the formalization of more complex statements compared to PL.

Linear Temporal Logic (LTL) [25] is a form of modal logic and a propositional logic framework designed to reason about the evolution of a finite set of propositional variables (e.g. p, q) over time. It achieves this by incorporating temporal operators (F for *finally*, G for *globally* and R for *release*, among others, which vary across fragments), which are defined in terms of the fundamental operators X for *next* and U for *until*, along with the logical operators inherited from PL. Unlike ATL, which is interval-based, LTL — and by extension FOLTL — is point-based: time is interpreted as a sequence of discrete time points rather than over intervals. This gives FOLTL a fundamentally different approach to temporal reasoning, fixing a semantics that is discrete and oriented along time points.

First-Order Linear Temporal Logic (FOLTL) [26, 32, 33] extends LTL with the expressive power of FOL, allowing predicates, variables, constants, functions, and first-order quantifiers to appear within temporal formulas. This integration enables the precise specification of time-dependent and data-rich behaviors in reactive and complex systems, where both temporal evolution and data structure play an essential role. Owing to its expressiveness and wide adoption in formal methods, FOLTL has become a suitable framework for the specification and verification of distributed and concurrent systems.

FOLTL has the greatest expressive power among all the logics presented here. Figure 2.3 illustrates the differences in expressive power between these logics across two distinct axes: higher-order features, i.e., the ability to quantify over entities, and temporal reasoning.

Figure 2.3: The increase in expressive power provided by FOLTL.



We do not present the full formal semantics of FOLTL here, as it depends on specific interpretation functions and environment definitions within a temporal model. For a detailed formal treatment, the reader is referred to [34]. Instead, we offer an intuitive overview of its temporal operators and illustrate their semantics through examples. This is intentional, as FOLTL is not used directly in our later evaluation but serves to illustrate the expressive capabilities of first-order temporal logic — particularly regarding quantification, predicates, and variables.

The following syntax of FOLTL is based on that presented by Geatti et al. [34].

Let $\Sigma = \mathcal{C} \cup \mathcal{V} \cup \mathcal{F} \cup \mathcal{P}$ be a first-order signature over mutually disjoint, finite sets of constants, variables, functions, and predicate symbols, respectively. The syntax of Σ-terms, which correspond to values, is defined as follows:

$$t := c \mid x \mid f(t_1, \ldots, t_n)$$

where $c \in \mathcal{C}$, $x \in \mathcal{V}$, $f \in \mathcal{F}$, and $t_1, \ldots, t_n$ are Σ-terms. The syntax of Σ-formulas, which evaluate to *true* or *false*, is defined as follows:

$$\phi := p(t_1, \ldots, t_n) \mid t_1 = t_2 \mid \neg\,\phi \mid \phi \vee \phi \mid \exists\, x\,.\,\phi \mid \mathsf{X}\,\phi \mid \mathsf{Y}\,\phi \mid \phi\ \mathsf{U}\ \phi \mid \phi\ \mathsf{S}\ \phi$$

where $p \in \mathcal{P}$, $t_1, \ldots, t_n$ are Σ-terms, $x \in \mathcal{V}$, and the temporal operators X, Y, U, and S are called next, yesterday, until and since. Derived constructs, such as $\wedge, \Rightarrow, \top, \forall$, F, O, G, H, R, and T are defined as usual.

Table 2.2 summarizes the FOLTL temporal operators, which underpin the illustrative semantics presented in Figure 2.4. The complete inductive formal semantics for boolean connectives, quantifiers, and temporal operators follow standard definitions (e.g., the one presented by Geatti et al. [34]).

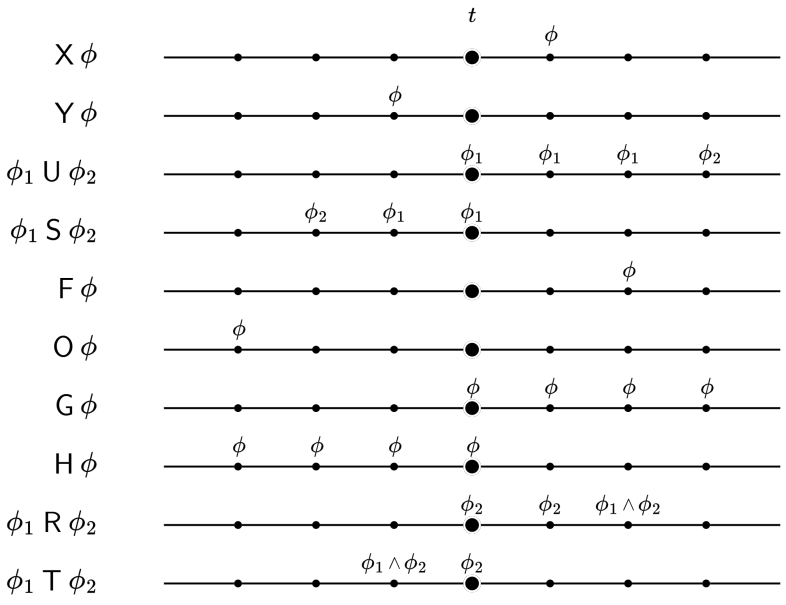
To give a broad and general idea of what can be expressed in FOLTL, some examples and their respective explanations are given in Figure 2.5.

This logic fragment inspired several key aspects of our own temporal logic design for reasoning about distributed protocols. In particular, we draw from FOLTL the notion of first-order reasoning, i.e., quantification over terms within a domain, and the use of predicates to describe system states and relationships. Additionally, we adopt its discrete conception of time as a basis for temporal reasoning,

Table 2.2: Summary of the semantics of FOLTL temporal operators.

Operator	Reads	Meaning
$\mathsf{X}\,\phi$	next	ϕ holds at the next time point
$\mathsf{Y}\,\phi$	yesterday	ϕ held at the previous time point
$\phi_1\,\mathsf{U}\,\phi_2$	until	ϕ_1 holds until ϕ_2 holds (and ϕ_2 eventually holds)
$\phi_1\,\mathsf{S}\,\phi_2$	since	ϕ_1 has held since ϕ_2 last held (and ϕ_2 held once)
$\mathsf{F}\,\phi$	eventually	ϕ will hold at some future time point
$\mathsf{O}\,\phi$	once	ϕ has held at some past time point
$\mathsf{G}\,\phi$	globally	ϕ holds at all future time points (including this one)
$\mathsf{H}\,\phi$	historically	ϕ has held at all past time points (including this one)
$\phi_1\,\mathsf{R}\,\phi_2$	release	ϕ_2 holds until (and including) a time point where ϕ_1 holds, if any exists
$\phi_1\,\mathsf{T}\,\phi_2$	triggered	ϕ_2 has held since (and including) a time point where ϕ_1 holds, if any exists

Figure 2.4: Illustration of the temporal operators of FOLTL at time point t.



while refraining from incorporating its temporal operators directly.

2.3 Two-Phase Commit

The Two-Phase Commit (2PC) protocol [1, 35] is a classic distributed consensus algorithm used to ensure atomic commitment in distributed database systems and transaction processing environments. Its purpose is to guaranty that all participating nodes in a distributed transaction either commit or abort the proposed transaction by maintaining global consistency even in the presence of failures. The 2PC protocol is particularly valuable in distributed systems where maintaining consistency across multiple

Figure 2.5: Examples of real-world scenarios specified using FOLTL.

$\exists\, n \,.\, \mathsf{G}\big(\forall\, country \,.\, \exists\, person \,.\, \textbf{head-of-state}(person, country)\big)$
It is always the case that for every country, there exists a person who is the head of state of that country.

$\forall\, person \,.\, \big(\big(\textbf{raining}() \wedge \neg\, \exists\, umbrella \,.\, \textbf{holding}(person, umbrella)\big) \Rightarrow \mathsf{F}\, \textbf{wet}(person)\big)$
For all people, if it is raining and there does not exist an umbrella that the person is holding, then that person will eventually become wet.

$\mathsf{G}\big(\textbf{email}(e) \wedge \textbf{person}(p)\big) \wedge \big(\textbf{being-written}(e, p)\ \mathsf{U}\ \big(\textbf{sent-by}(e, p) \wedge \mathsf{X}\big(\textbf{delivered}(e) \vee \textbf{failed}(e)\big)\big)\big)$
There exists an email and a person, both of which remain valid entities, such that the person continues writing the email until it is sent, after which it is either successfully delivered or fails to send.

autonomous nodes is critical, such as in cloud transaction systems, blockchain consensus layers, and distributed databases [36].

The protocol operates under a coordinator–participant model and proceeds in two distinct phases: a voting phase and a commit phase. In the first phase, the coordinator sends a prepare request to all participants, asking whether they are ready to commit the transaction. Each participant performs local checks (e.g., data integrity, resource availability) and responds with either a commit vote or abort vote message. In the second phase, based on the collected votes, the coordinator decides whether to commit or abort the transaction. Unlike most consensus algorithms, which rely on majority voting, 2PC requires unanimous agreement: if all participants vote to commit, the coordinator sends a global commit instruction; otherwise, it sends a global abort instruction. These phases ensure both atomicity — all participants reach the same decision —and durability, as committed changes remain permanent even in the presence of subsequent failures.

Although conceptually simple, 2PC must handle failures and asynchronous communication, and it is known to suffer from blocking in the presence of coordinator failures. This limitation motivated later refinements such as the Three-Phase Commit (3PC) protocol [37] and more general consensus mechanisms, like Paxos [38].

The following formalization introduces the main operations of the 2PC protocol and specifies several key correctness properties — including eventual decision, atomic commitment, and commit justification — using FOLTL, as an example of how temporal logics can express distributed system behavior. For this purpose, our model assumes a single coordinator and multiple uniquely identified participant nodes. Upon receiving a prepare request, a participant writes the transaction data to disk, checks database consistency and constraints, and, if valid, votes to commit, locking the necessary resources. The example correctness properties are modeled as FOLTL formulas using predicates parameterized by nodes n, transactions t, and data items d. In the specifications that follow, a dash ($-$) denotes a wildcard

argument — a parameter not bound by quantification and free to take any value in its domain.

- $\mathbf{request}(n, t, d)$: the coordinator requests node n to prepare for transaction t with data d.
- $\mathbf{v\text{-}commit}(n, t)$: node n votes to commit transaction t, sending the response to the coordinator.
- $\mathbf{v\text{-}abort}(n, t)$: node n votes not to commit transaction t, sending the response to the coordinator.
- $\mathbf{i\text{-}commit}(t)$: the coordinator instructs all participant nodes to commit transaction t.
- $\mathbf{i\text{-}abort}(t)$: the coordinator instructs all participant nodes not to commit transaction t.
- $\mathbf{e\text{-}commit}(n, t)$: node n executes the commit for transaction t, applying the pending data changes.
- $\mathbf{e\text{-}abort}(n, t)$: node n executes the abort for transaction t, discarding pending data changes.

(LIVENESS) **Eventual decision**: if the coordinator requests any participant node n to prepare for any transaction t, all participant nodes will eventually be instructed to either commit or abort that transaction.

$$\forall\, n\,.\,\forall\, t\,.\,\mathsf{G}\Big(\mathbf{request}(n, t, -) \Rightarrow \mathsf{F}\big(\mathbf{i\text{-}commit}(t) \vee \mathbf{i\text{-}abort}(t)\big)\Big)$$

(SAFETY) **Atomic commitment**: if the coordinator instructs all participant nodes to commit any transaction t, then all nodes n must have voted to commit that transaction; if it instructs to abort t, then at least one node n must have voted to abort it.

$$\forall\, t\,.\,\mathsf{G}\Big(\big(\mathbf{i\text{-}commit}(t) \Rightarrow \forall\, n\,.\,\mathsf{O}\,\mathbf{v\text{-}commit}(n, t)\big) \wedge \big(\mathbf{i\text{-}abort}(t) \Rightarrow \exists\, n\,.\,\mathsf{O}\,\mathbf{v\text{-}abort}(n, t)\big)\Big)$$

(SAFETY) **Commit justification**: if any node n executes a commit of any transaction t, then the coordinator must have instructed all participant nodes to commit that transaction, which must have been preceded by a vote to commit from that node and, in turn, a prepare request.

$$\forall\, n\,.\,\forall\, t\,.\,\mathsf{G}\Big(\mathbf{e\text{-}commit}(n, t) \Rightarrow \mathsf{O}\big(\mathbf{i\text{-}commit}(t) \wedge \mathsf{O}(\mathbf{v\text{-}commit}(n, t) \wedge \mathsf{O}\,\mathbf{request}(n, t, -))\big)\Big)$$

These same operations and properties are later reformulated using the formal frameworks introduced in this work: dFOATL in Section 4.2.1 and the ACTL fragments in Section 5.3.1.

2.4 Distributed Hash Table

A DHT protocol [1, 23] is a foundational distributed algorithm that implements a scalable, decentralized key–value store across a peer-to-peer network. Its goal is to insert and retrieve $(\text{key}, \text{value})$ pairs without a central server while providing logarithmic-time lookups and balanced load distribution, making DHTs

useful for content distribution, peer-to-peer systems, decentralized storage, and blockchain networks. Representative DHTs include Pastry [39], Kademlia [40], and Chord [23], an early DHT that arranges nodes in a ring and maintains it through stabilization to handle joins, leaves, and failures.

A DHT handles client requests (store, lookup, find node, remove) by routing them along overlay links according to the protocol's specific routing metric, which varies by implementation. Store operations replicate key–value pairs to responsible nodes, lookup and find node requests reach the node holding the key, and remove deletes the pair. Background maintenance operations, such as join, leave, stabilization, and replication repair, ensure correctness under churn and affect whether lookups return the latest values.

Although highly scalable, DHTs must cope with frequent joins and leaves, network partitions, and failures while maintaining availability and reasonable consistency. Different designs balance lookup efficiency, fault tolerance, and network overhead; protocol-specific enhancements, such as Kademlia's XOR metric and Pastry's proximity-aware routing, improve lookup robustness and latency.

In our model, the network consists of uniquely identified nodes that store key–value pairs. While network states span multiple time points, operations take inputs and yield outputs only upon completion. Thus, for this DHT scenario, we model them as instantaneous, with explicit FOLTL predicates marking their beginning and end. Effects of operations are defined to occur between these events. Predicates are parameterized by nodes n, keys k, values v, and unique identifiers z that link related predicates. For readability, these identifiers appear as subscripts but are semantically just additional arguments. Examples of operations, network states, and properties include:

- $\mathbf{b\text{-}store}_z(n, k, v)$: the client requests node n to store the key-value pair (k, v) in the DHT.
- $\mathbf{e\text{-}store}_z(n')$: the operation completes at node n', which notifies the client upon completion.
- $\mathbf{b\text{-}lookup}_z(n, k)$: the client requests the value for key k from node n in the DHT.
- $\mathbf{e\text{-}lookup}_z(n', v)$: the operation completes at the node n' storing (k, v) and returns v to the client.
- $\mathbf{stable}$: the network remains unchanged, with no node joins, departures, or failures.

(SAFETY) **Lookup consistency**: if a lookup z obtains a value v for a given key k, then the (k, v) pair was previously written by a store operation and there are no fabricated values.

$$\forall\, k, v\,.\, \mathsf{G}\Big(\forall\, z\,.\,\big(\mathbf{b\text{-}lookup}_z(-, k) \wedge \mathsf{F}\,\mathbf{e\text{-}lookup}_z(-, v)\big) \Rightarrow \mathsf{O}\,\mathbf{b\text{-}store}(-, k, v)\Big)$$

(SAFETY) **Value freshness**: if a lookup z obtains a value v for a given key k, then v is the latest stored value for key k, that is, any past store of a different value v' for k must have been followed by a store of v.

$$\forall\, k, v\,.\, \mathsf{G}\Big(\forall\, z\,.\,\big(\,\mathbf{b\text{-}lookup}_z(-, k) \wedge \mathsf{F}\,\mathbf{e\text{-}lookup}_z(-, v)\big) \Rightarrow \big(\forall\, v' \neq v\,.\,\mathsf{O}\big(\mathbf{b\text{-}store}(-, k, v') \Rightarrow \mathsf{F}\,\mathbf{b\text{-}store}(-, k, v)\big)\big)\Big)$$

(LIVENESS) **Operation termination**: if the network is stable, then every store or lookup operation eventually completes, that is, for any operation z, once it starts, it will eventually end.

$$\forall\, z \,.\, \mathsf{G}\Big(\mathbf{stable} \Rightarrow \big(\big(\mathbf{b\text{-}store}_z(\ldots) \Rightarrow \mathsf{F}\, \mathbf{e\text{-}store}_z(-)\big) \wedge \big(\mathbf{b\text{-}lookup}_z(\ldots) \Rightarrow \mathsf{F}\, \mathbf{e\text{-}lookup}_z(\ldots)\big)\big)\Big)$$

In these property specifications, operations are represented by paired begin and end events rather than single predicates spanning multiple time points. While more verbose, this approach better captures client–server interactions, clarifies input/output distinctions, and contrasts with the 2PC example, highlighting FOLTL's flexibility in supporting different specification styles.

While FOLTL enables the expression of arbitrary properties, syntactically tailored fragments may simplify specifications for DHTs and similar protocols, where operations are continuous and inputs and outputs are clearly separated.

To address this need, we introduce in later chapters two dedicated formalisms that build on the notion of intervals and borrow relations from ATL to more naturally express and verify such behaviors. Examples of DHT operations and properties specified using dFOATL are presented in Section 4.2.2, and using ACTL are provided in Section 5.3.2.

3

Related Work

Contents

This chapter reviews prior research on the specification and testing of distributed systems. It begins with an overview of temporal reasoning logics in Section 3.1, focusing on ATL and FOLTL, their real-world applications, limitations, tractable fragments, and tool support. It then examines state-of-the-art approaches for specifying distributed protocols in Section 3.2, highlighting the use of TLA+ and Alloy for modeling and verifying protocol correctness through trace analysis and model-based validation. Finally, Section 3.3 reviews formal testing efforts on DHTs, including studies on Pastry and Chord that uncovered flaws and motivated recent work on ATL-based axiomatizations for automated trace validation. Collectively, these works trace the evolution from formal specification to practical, tool-assisted testing of distributed protocols.

3.1 Time Reasoning Logics

There are several logics for reasoning about time-spanning operations and their interrelations (e.g., ATL [24], LTL [25], FOLTL [26], TLA [20], and EC [27]). This section studies the applications to real-world scenarios, work done, and drawbacks of ATL and FOLTL, on Section 3.1.1 and Section 3.1.2, as these two prove to be strong candidates for general-purpose specification and testing of complex systems.

3.1.1 Allen Temporal Logic

ATL [24] provides a formalism for representing qualitative temporal relations between intervals.

In the literature, reasoning about such temporal relations is often formulated as a Constraint Satisfaction Problem (CSP), with constraint propagation algorithms commonly used in practice [41]. This classic approach underpins many applied systems, including Natural Language Processing (NLP) applications [42] and scheduling problems [43].

Allen's interval relations have been applied as part of frameworks for reasoning about temporal relationships among clinical events, supporting tasks such as patient cohort identification in healthcare data [44]. Similarly, they have been used in frameworks to model temporal relationships among events in video sequences, enabling hierarchical event detection and analysis [45].

Research has investigated which fragments of ATL are computationally practical, focusing on tractable subsets that allow efficient reasoning [46].

Until recently, despite its significant impact in computer science, no systems were available for the automatic verification of ATL specifications and no model-checking technique existed for ATL. Roşu and Bensalem [31] were the first to address this limitation by proposing an encoding of ATL formulas into LTL [25]. Their approach required restricting ATL to discrete time, interpreting intervals over the natural numbers ($i = [t_1, t_2] \subseteq \mathbb{N}$). However, this encoding had limitations: it lacked an implementation and used nested temporal operators, which negatively impact performance.

Later, implemented using the Alloy Analyzer [21], Policarpo et al. proposed an encoding of ATL into FOLTL [22]. Similarly, Janhunen and Sioutis [47] present an efficient encoding of Allen's interval relations in Answer Set Programming (ASP) with difference constraints, enabling scalable temporal reasoning for planning, scheduling, and healthcare applications.

3.1.2 First-Order Linear Temporal Logic

FOLTL [26] extends FOL with temporal operators to reason about how properties involving objects and relations change over linear time.

FOLTL often relies on automata-theoretic methods as a foundation, translating formulas into automata over infinite traces (e.g., Büchi automata) where model satisfaction corresponds to automaton acceptance [48]. Alongside tableau-based reasoning and Boolean Satisfiability (SAT)/Satisfiability Modulo Theories (SMT) solvers, these methods underpin model checking [49] — both bounded and unbounded — as well as runtime verification, to verify whether a model satisfies a given specification. FOLTL also supports theorem proving through symbolic, logic-based reasoning [50].

FOLTL has been applied to real-world monitoring and verification. MonPoly combines first-order and metric temporal logic to monitor policy, compliance, and security logs with expressive parametric formulas [51]. JavaMOP enables instrumentation-based monitoring of Java programs, detecting Application Programming Interface (API) misuse, data races, and runtime defects by translating parametric temporal properties into bytecode [52]. FOLTL has also been used to analyze data-aware processes and database systems, as in linDMT [53].

Expressive power often comes at the cost of decidability. When satisfiability checking is required (e.g., for online monitoring or model checking), the decidability of FOLTL must be considered. Full FOLTL is highly undecidable, as it combines first-order logic with temporal reasoning; even simple fragments can be non-recursively enumerable [54].

Nonetheless, certain less expressive fragments are tractable or (semi-)decidable, and recent work has explored their use for online monitoring [55–57]. Fragments of the sort are implemented in tools like BLACK [58], which leverages SAT/SMT solving, and analyzers like Alloy [21], which can also be adapted for checking properties at runtime as the system executes by encoding specifications into their languages.

3.2 Specifying Distributed Protocols

Specification-based testing approaches for distributed protocols have long been used to detect violations of desired properties. These approaches can rely on temporal logic formalisms, as well as on process calculi, automata models, or state machines. They use formal specification languages to model expected behaviors, such as TLA+ [12], which has been the basis for most work on log validation in distributed systems. In this context, specifications are verified against traces collected from instrumented applications using the TLC model checker.

TLA+ has been employed to prevent bugs in AWS protocols from reaching production [59] and to detect violations in protocol implementations by checking whether execution traces comply with a specification after execution [18]. Similarly, Alloy [21] has been used to formally specify the Chord protocol [23], with the Alloy Analyzer exploring these specifications to identify potential counterexamples [14]. Using the same model finder, later work validated increasingly larger execution traces of the OpenChord proto-

col against property specifications expressed in fragments of ATL, in an offline, model-based monitoring setting [22].

Howard et al. [60] introduced the model-based trace checking approach for validating distributed software, instrumenting applications to log executions and verifying LTL specifications by translating logs into Promela models checked with Spin [61]. A comparative study of Alloy and Spin evaluated their applicability to protocol verification, revealing unexpected insights and offering recommendations for choosing formal methods [62].

3.3 Testing Distributed Hash Tables

Pastry is a well-known DHT protocol. Lu et al. [63] tested its lookup operations using a TLA+ model, revealing that concurrent joins could compromise correctness. To address this, they proposed a modification restricting each node to handle a single join at a time. This result was later reinforced by Azmy et al. [15], who provided a formal proof of the property in a simplified join-only model.

As mentioned in the previous section, Alloy has also been employed to model Chord, another prominent DHT protocol. Through formal analysis, Zave [14, 64] uncovered violations in Chord's original ring-maintenance invariants [65] and proposed corresponding corrections. She further introduced a global invariant ensuring eventual convergence of the network to its ideal state.

These counterexamples found in implementations of DHTs highlight the need for systematic testing frameworks tailored to distributed protocols through trace validation. Our motivation to use DHTs as a case study for developing and evaluating the performance of such a monitoring framework is inspired by recent work by Policarpo et al., which is yet not publicly accessible but extends their previous preparation work [22].

In that work, the authors present the first implementation-independent axiomatization of DHTs, using an evolution of ATL, that focuses on functional rather than topological properties. They introduce a semi-automatic testing pipeline for validating DHT implementations against this axiomatization. Although currently centered on DHTs, their pipeline can be extended to other distributed protocols. The framework tests eight key properties — value, key, and structural — against progressively larger logs instrumented from OpenChord executions, using an Alloy model where ATL relations are encoded into FOLTL.

Although this work is relevant, we consider the use of Alloy for trace monitoring against distributed protocol specifications suboptimal, as it was not designed for this purpose and does not scale well, as the authors themselves acknowledge. The Alloy-based monitor performs efficiently for logs up to about 250 entries, but beyond that, evaluation time increases sharply and memory exhaustion may occur, especially for properties with deeply nested quantifiers. To overcome these limitations, we propose an alternative tool specifically designed for this task, minimizing overhead and achieving greater efficiency.

4

Discrete First-Order Allen Temporal Logic

Contents

In this chapter, we introduce a discrete first-order variant of ATL, called dFOATL. This logic combines the strengths of ATL and FOLTL, addressing their respective limitations. While ATL effectively captures temporal relations between intervals but lacks first-order expressiveness, FOLTL supports rich data reasoning with quantification but is syntactically cumbersome and less intuitive for modeling protocol-level events. dFOATL integrates these advantages to provide a more natural and expressive framework for specifying and monitoring distributed protocols. The formal syntax and semantics are presented in Section 4.1. To demonstrate its expressiveness, we provide examples in Section 4.2, illustrating how dFOATL can specify implementation-independent properties of two distributed protocols. In Section 4.3,

we present the decision procedure for dFOATL together with a corresponding analysis. Finally, Section 4.4 discusses the limitations of this fragment and outlines the motivation for exploring a new, more expressive fragment.

4.1 Syntax and Semantics

In this section, we first present the definition of the syntax of dFOATL formulas in Section 4.1.1, illustrating how Allen's interval relations are combined with first-order constructs. Subsequently, in Section 4.1.2, we formalize the semantics of the proposed dFOATL fragment, making use of the introduced environment mappings.

4.1.1 Syntax

Let $\mathcal{V}$ be a set of finite domain variable symbols or constants, $\mathcal{I}$ a set of time interval variable symbols, $\mathcal{P}$ a set of predicate symbols involving domain variables, and $\mathcal{R}$ a set of Allen's relation symbols between interval variables, where these sets are all finite and mutually disjoint.

Formulas of dFOATL are denoted by ϕ, and their syntax is defined as follows:

$$
\begin{aligned}
\phi \ ::= \ & \neg\,\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \Rightarrow \phi_2 \\
& \mid \boxed{\forall\, x\,.\,\phi} \mid \boxed{\exists\, x\,.\,\phi} \mid \boxed{\forall\, i\,.\,\phi} \mid \boxed{\exists\, i\,.\,\phi} \\
& \mid \boxed{\mathit{Holds}(p(\overline{x}), i)} \mid \boxed{\mathit{Occurs}(p(\overline{x}), i)} \\
& \mid x_1 = x_2 \mid r(i_1, i_2)
\end{aligned}
$$

where $x \in \mathcal{V}$ is a domain variable symbol or a constant, $i \in \mathcal{I}$ is an interval variable symbol, $p \in \mathcal{P}$ is a predicate symbol, $\overline{x}$ is a finite sequence of domain variable symbols or constants $x_1, \ldots, x_{|p|}$ where $x_n \in \mathcal{V}$ (for any $n \in \{1, \ldots, |p|\}$), and $r \in \mathcal{R}$ is a relation symbol. We distinguish between quantifiers over domain variables and interval variables to enhance the clarity of specifications. Unlike in ATL, propositional relations in dFOATL may include predicates.

The syntax of Allen's interval relations is defined as follows:

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

where the inverse interval relations — $\mathit{After}(i_1, i_2)$, $\mathit{MetBy}(i_1, i_2)$, $\mathit{OverlappedBy}(i_1, i_2)$, $\mathit{StartedBy}(i_1, i_2)$, $\mathit{Contains}(i_1, i_2)$, and $\mathit{FinishedBy}(i_1, i_2)$ — have been omitted and will not be considered in this work for

the sake of simplicity. Note that these syntactic definitions are agnostic to whether time is discrete or continuous, as this distinction pertains to the semantics, as is the case with classic ATL.

With a slight abuse of notation, we may adopt the following quantification of domain variables shorthand from now on for improved clarity and consistency, by representing a sequence of nested individual quantifiers:

$$\forall \overline{x}, \overline{y} \equiv \forall x_1 \ldots \forall x_n \,.\, \forall y_1 \ldots \forall y_m$$

4.1.2 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), where t_b and t_e are discrete time points belonging to the set of natural numbers, $\mathbb{N}$, representing, respectively, the beginning and end of the interval:

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

In the case where x is a constant, $\sigma(x) = x$.

Let ω be the predicate environment, i.e., the finite mapping from predicate symbols $p \in \mathcal{P}$ and the values of their parameters $\sigma(x_n)$ in D_{x_n} (for all $n \in \{1, \ldots, |p|\}$), to the set of maximal pairs of time points where $p(\sigma(\overline{x}))$ is true:

$$\omega : \mathcal{P} \times D^{|p|} \to \wp(\mathbb{N} \times \mathbb{N})$$

The verification of a dFOATL formula ϕ with respect to predicate and variable environments ω and σ, denoted as $\omega, \sigma \models_{\text{dFOATL}} \phi$, is defined inductively as follows:

1. $\omega, \sigma \models_{\text{dFOATL}} \neg\phi \iff \textbf{not}\ \omega, \sigma \models_{\text{dFOATL}} \phi$
2. $\omega, \sigma \models_{\text{dFOATL}} \phi_1 \vee \phi_2 \iff \omega, \sigma \models_{\text{dFOATL}} \phi_1\ \textbf{or}\ \omega, \sigma \models_{\text{dFOATL}} \phi_2$
3. $\omega, \sigma \models_{\text{dFOATL}} \phi_1 \wedge \phi_2 \iff \omega, \sigma \models_{\text{dFOATL}} \phi_1\ \textbf{and}\ \omega, \sigma \models_{\text{dFOATL}} \phi_2$
4. $\omega, \sigma \models_{\text{dFOATL}} \phi_1 \Rightarrow \phi_2 \iff \omega, \sigma \models_{\text{dFOATL}} \phi_1\ \textbf{implies}\ \omega, \sigma \models_{\text{dFOATL}} \phi_2$
5. $\omega, \sigma \models_{\text{dFOATL}} \forall x \,.\, \phi \iff \textbf{forall}\ v \in D_x \,.\, \omega, \sigma[x \mapsto v] \models_{\text{dFOATL}} \phi$
6. $\omega, \sigma \models_{\text{dFOATL}} \exists x \,.\, \phi \iff \textbf{exists}\ v \in D_x \,.\, \omega, \sigma[x \mapsto v] \models_{\text{dFOATL}} \phi$
7. $\omega, \sigma \models_{\text{dFOATL}} \forall i \,.\, \phi \iff \textbf{forall}\ t_b, t_e \in \mathbb{N} \,.\, \omega, \sigma[i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} \phi$
8. $\omega, \sigma \models_{\text{dFOATL}} \exists i \,.\, \phi \iff \textbf{exists}\ t_b, t_e \in \mathbb{N} \,.\, \omega, \sigma[i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} \phi$

9. $\omega, \sigma \models_{\text{dFOATL}} Holds(p(\overline{x}), i) \iff \textbf{exists}\ \overline{v} \in D^{|p|},\ t_b, t_e, t_{b'}, t_{e'} \in \mathbb{N}\ .$
$\sigma(\overline{x}) = \overline{v} \wedge \sigma(i) = (t_b, t_e) \wedge \omega(p, \overline{v}) \ni (t_{b'}, t_{e'}) \wedge t_{b'} \leq t_b < t_e \leq t_{e'}$

10. $\omega, \sigma \models_{\text{dFOATL}} Occurs(p(\overline{x}), i) \iff \textbf{exists}\ \overline{v} \in D^{|p|},\ t_b, t_e, t_{b'}, t_{e'} \in \mathbb{N}\ .$
$\sigma(\overline{x}) = \overline{v} \wedge \sigma(i) = (t_b, t_e) \wedge \omega(p, \overline{v}) \ni (t_{b'}, t_{e'}) \wedge t_b \leq t_{e'} \wedge t_{b'} \leq t_e$

11. $\omega, \sigma \models_{\text{dFOATL}} x_1 = x_2 \iff \sigma(x_1) = \sigma(x_2)$

12. $\omega, \sigma \models_{\text{dFOATL}} r(i_1, i_2) \iff [\![r]\!](\sigma(i_1), \sigma(i_2))$

As defined in Item 7 and Item 8, the quantifiers $\forall$ and $\exists$ are introduced to clearly distinguish quantification over interval variables from the domain variable quantification expressed by $\forall$ and $\exists$ in Item 5 and Item 6, respectively.

Informally, Item 9 states that the relation $Holds(p(\overline{x}), i)$ is true when the predicate $p(\overline{x})$ holds throughout the entirety of the interval i, whereas Item 10 states that $Occurs(p(\overline{x}), i)$ is true if the predicate $p(\overline{x})$ holds at least once during the interval i.

For clarity and brevity, we have adopted a slight abuse of notation, abbreviating point-wise evaluation of domain variable symbols in the variable environment σ to sequences:

$$\sigma(\overline{x}) = \overline{v} \ \equiv \ \bigwedge_{n=1}^{|\overline{x}|} \sigma(x_n) = v_n$$

The semantics of Allen's interval relations is defined formally as follows:

$$[\![Before]\!]((t_{b_1}, t_{e_1}), (t_{b_2}, t_{e_2})) \iff t_{e_1} < t_{b_2}$$

$$[\![Meets]\!]((t_{b_1}, t_{e_1}), (t_{b_2}, t_{e_2})) \iff t_{e_1} = t_{b_2}$$

$$[\![Overlaps]\!]((t_{b_1}, t_{e_1}), (t_{b_2}, t_{e_2})) \iff t_{b_1} < t_{b_2} < t_{e_1} < t_{e_2}$$

$$[\![Starts]\!]((t_{b_1}, t_{e_1}), (t_{b_2}, t_{e_2})) \iff t_{b_1} = t_{b_2} \wedge t_{e_1} < t_{e_2}$$

$$[\![During]\!]((t_{b_1}, t_{e_1}), (t_{b_2}, t_{e_2})) \iff t_{b_2} < t_{b_1} \wedge t_{e_1} < t_{e_2}$$

$$[\![Finishes]\!]((t_{b_1}, t_{e_1}), (t_{b_2}, t_{e_2})) \iff t_{e_1} = t_{e_2} \wedge t_{b_2} < t_{b_1}$$

$$[\![Equals]\!]((t_{b_1}, t_{e_1}), (t_{b_2}, t_{e_2})) \iff t_{b_1} = t_{b_2} \wedge t_{e_1} = t_{e_2}$$

4.2 Specification of Distributed Protocols

This section demonstrates the capabilities of the dFOATL framework by specifying the operations and correctness properties of representative distributed protocols. The 2PC protocol is presented in Section 4.2.1, followed by the DHT protocol in Section 4.2.2. In these property specifications, the wildcard

symbol $(-)$ indicates a parameter that is irrelevant within nested quantifiers or to the property being expressed.

4.2.1 Two-Phase Commit

The Two-Phase Commit (2PC) protocol was introduced in Section 2.3. We assume the same model, consisting of a single coordinator and multiple uniquely identified participant nodes. For details on the protocol's operation, the reader is referred to that section.

In this formalization, the operations of the 2PC protocol are expressed using dFOATL predicates $p(\overline{x})$, similar to the FOLTL-based specifications.

The predicates are parameterized by nodes n, transactions t, data items d, and vote outcomes v, where v takes one of the constants commit or abort — as an exploratory step, we generalized the vote into a parameter, effectively merging some of the previously distinct operations.

The correctness properties from Section 2.3 are then reformulated as dFOATL formulas, making use of $Occurs(p(\overline{x}), i)$ relations which extend predicates over an interval, instead of being point-wise. This offers a clear basis for comparing the two specification styles.

- $\mathbf{request}(n, t, d)$: the coordinator requests node n to prepare for transaction t with data d.
- $\mathbf{vote}(n, v, t)$: node n votes to commit / abort transaction t, sending the response to the coordinator.
- $\mathbf{instruct}(v, t)$: the coordinator instructs all participant nodes to either commit / abort transaction t.
- $\mathbf{execute}(n, v, t)$: node n executes the commit or abort for transaction t, applying or discarding the pending data changes, respectively.

(LIVENESS) **Eventual decision**: if the coordinator requests any participant node n to prepare for any transaction t, all participant nodes will eventually be instructed to either commit or abort that transaction.

$$\forall\, n, t\,.\, \forall\!\!\!/\, i_1\,.\, Occurs(\mathbf{request}(n, t, -), i_1) \Rightarrow \exists\!\!\!/\, i_2\,.\, Occurs(\mathbf{instruct}(-, t), i_2) \wedge Before(i_1, i_2)$$

(SAFETY) **Atomic commitment**: if the coordinator instructs all participant nodes to commit any transaction t, then all nodes n must have voted to commit that transaction; if it instructs to abort t, then at least one node n must have voted to abort it.

$$\begin{aligned}\forall\, v_1, t\,.\, \forall\!\!\!/\, i_1\,.\, Occurs(\mathbf{instruct}(v_1, t), i_1) \Rightarrow \\ \big(v_1 = \mathrm{commit} \wedge \forall\, n, v_2\,.\, \forall\!\!\!/\, i_2\,.\, Occurs(\mathbf{vote}(n, v_2, t), i_2) \Rightarrow Before(i_2, i_1) \wedge v_1 = v_2\big) \\ \vee\, \big(v_1 = \mathrm{abort} \wedge \exists\!\!\!/\, n\,.\, \exists\!\!\!/\, i_2\,.\, Occurs(\mathbf{vote}(n, v_1, t), i_2) \wedge Before(i_2, i_1)\big)\end{aligned}$$

(SAFETY) **Commit justification**: if any node n executes a commit of any transaction t, then the coordinator must have instructed all participant nodes to commit that transaction, which must have been preceded by a vote to commit from that node and, in turn, a prepare request.

$$\begin{aligned}\forall\, n, t\,.\,\forall\, i_1\,.\, \mathit{Occurs}(\mathbf{execute}(n, \mathrm{commit}, t), i_1) \Rightarrow \exists\, i_2\,.\, \mathit{Occurs}(\mathbf{instruct}(\mathrm{commit}, t), i_2) \wedge \mathit{Before}(i_2, i_1)\\ \wedge\, \exists\, i_3\,.\, \mathit{Occurs}(\mathbf{vote}(n, \mathrm{commit}, t), i_3) \wedge \mathit{Before}(i_3, i_2)\\ \wedge\, \exists\, i_4\,.\, \mathit{Occurs}(\mathbf{request}(n, t, -), i_4) \wedge \mathit{Before}(i_4, i_3)\end{aligned}$$

4.2.2 Distributed Hash Table

The Distributed Hash Table (DHT) protocol was introduced in Section 2.4. We assume the same network model, consisting of uniquely identified nodes that store key–value pairs. For details on the protocol's operation, the reader is referred to that section.

In this formalization, the protocol's operations are described using dFOATL predicates, which are parameterized by nodes n, keys k and values v. The correctness properties from Section 2.4 are reformulated as dFOATL formulas using relations $\mathit{Occurs}(p(\overline{x}), i)$ to assign intervals to the predicates.

- $\mathbf{store}(n, k, v, n')$: 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.
- $\mathbf{lookup}(n, k, n', v)$: the client requests the value for key k from node n in the DHT; the operation completes at the node n' storing (k, v) and returns v to the client.

Unlike the 2PC example, we do not assume an extension of discrete-time operations to interval-based ones. Instead, we merge their beginning and end and combine their input and output parameters within a single predicate, as follows in Table 4.1. This reflects our motivation for employing intervals to more naturally capture distributed operations that unfold over multiple time points.

Table 4.1: Simplification iterative process in the definition of DHT operations.

FOLTL	dFOATL
$\mathbf{b\text{-}store}_z(n, k, v)$	$\mathbf{store}(n, k, v, n')$
$\mathbf{e\text{-}store}_z(n')$	
$\mathbf{b\text{-}lookup}_z(n, k)$	$\mathbf{lookup}(n, k, n', v)$
$\mathbf{e\text{-}lookup}_z(n', v)$	

(SAFETY) **Lookup consistency**: if a lookup obtains a value v for a given key k, then the (k, v) pair was previously written by a store operation and there are no fabricated values.

$$\forall\, k, v\,.\,\forall\, i_1\,.\, \mathit{Occurs}(\mathbf{lookup}(-, k, -, v), i_1) \Rightarrow \exists\, i_2\,.\, \mathit{Occurs}(\mathbf{store}(-, k, v, -), i_2) \wedge \mathit{Before}(i_2, i_1)$$

(SAFETY) **Value freshness**: if a lookup obtains a value v for a given key k, then v is the latest stored value for key k, that is, any past store of a different value v' for k must have been followed by a store of v.

$$\begin{aligned}
&\forall\, k, v_1 \,.\, \forall\!\!\!/\, i_1 \,.\, \mathit{Occurs}(\mathbf{lookup}(-, k, -, v_1), i_1) \Rightarrow \\
&\quad \big(\forall\, v_2 \,.\, \forall\!\!\!/\, i_2 \,.\, \mathit{Occurs}(\mathbf{store}(-, k, v_2, -), i_2) \wedge \mathit{Before}(i_2, i_1) \wedge v_1 \neq v_2 \Rightarrow \\
&\qquad \exists\!\!\!/\, i_3 \,.\, \mathit{Occurs}(\mathbf{store}(-, k, v_1, -), i_3) \wedge \mathit{Before}(i_2, i_3) \wedge \mathit{Before}(i_3, i_1)\big)
\end{aligned}$$

Note that the operation termination property is omitted: since DHT operations are modeled as single interval predicates rather than separate begin/end point-wise ones, that notion of termination is no longer meaningful.

4.3 Decision Procedure

This section introduces a decision procedure for verifying dFOATL formulas given variable and predicate environments. The latter, denoted by ω, encodes all relevant information of a trace and thus effectively serves as one, making it suitable for the offline monitoring task proposed. The procedure is defined as a recursive algorithm that mirrors the syntactic structure of dFOATL, ensuring that each construct is evaluated according to its formal semantics. The pseudocode of the algorithm is presented in Section 4.3.1, and its worst-case computational complexity is analyzed in Section 4.3.2.

4.3.1 Pseudocode

The pseudocode in Algorithm 4.1 defines the CHECK(ω, σ, ϕ) function, which returns *true* if and only if the formula ϕ is verified by the predicate environment ω under the variable environment σ. The algorithm handles all syntactic forms of dFOATL formulas.

4.3.2 Complexity Analysis

When verifying the relations $\mathit{Holds}(p(\overline{x}), i)$ and $\mathit{Occurs}(p(\overline{x}), i)$, the complexity is $\mathcal{O}(r)$, where r denotes the maximum number of pairs of time points to be checked in the set $\omega(p, \sigma(\overline{x}))$. In the case where at most one pair occurs per set ($r = 1$), this reduces to constant time, $\mathcal{O}(1)$.

The main source of computational complexity in the predicate environment verification algorithm comes from the variable quantifiers. Each domain variable quantifier naively explores all possible values within the finite domain of the variable, resulting in a worst-case complexity of $\mathcal{O}(d)$, where d is the size of the largest variable domain. Interval variable quantifiers, in turn, must consider all possible pairs of time

points within the bounded subset of $\mathbb{N}$, which leads to a complexity of $\mathcal{O}(n^2)$, assuming that checking each time point is linear in n.

Let q denote the total number of nested quantifiers in the formula being verified. Because the checking procedure is recursive, the overall worst-case complexity becomes $\mathcal{O}(\max(d, n^2)^q)$.

Because n^2 can be treated as just another finite domain and, by definition, d is the size of the largest domain, d can encompass both upper bounds, so $\max(d, n^2) = d$. Furthermore, the number of nested quantifiers is proportional to the syntactic size of the formula ($q = |\varphi|$).

$$\textsc{Check}(\omega, \sigma, \phi) \in \mathcal{O}(r \cdot d^{|\varphi|})$$

The verification problem for dFOATL over predicate environments is decidable but computationally expensive, exhibiting exponential time complexity in the number of nested quantifiers. This places it, at minimum, in the class of PSPACE-hard problems and, more generally, EXPTIME-complete, consistent with the known complexity of related first-order temporal logics.

In practice, interval variable quantifiers do not explore all possible combinations of t_b and t_e: for each t_b, only greater values of t_e are considered.

The domains of the domain variables are specific to the distributed protocol being modeled. However, they are always assumed to be finite, as is appropriate for our offline monitoring setting of complete traces. Similarly, the number of time points considered for interval quantification is bounded. If all these are large, the algorithm may not scale well.

This decision procedure is naive and not optimal, as the search performed by the variable quantifiers is blind. That is, all possible values within each domain are explored before the predicates themselves are evaluated. A more efficient approach would leverage the prior knowledge available in the predicate environment, which already encodes the intervals where instances of operations with corresponding parameter values occur, and thus restrict the exploration to the observable values relevant to each operation.

4.4 Thoughts on dFOATL

Although the current dFOATL approach is interesting and represents a step in the right direction, we identify room for further refinement.

In basic propositional ATL, expressing a distributed operation requires either associating the interval i directly with an action or treating a proposition p as the action occurring within an interval. In both cases, we can reason about temporal dependencies but not about data dependencies among operations.

By contrast, dFOATL allows for the explicit representation of both temporal and data relationships. However, it requires expressing operations as predicates $p(x_1, \ldots, x_n)$ embedded within $Holds(p(\overline{x}, i))$

or $Occurs(p(\overline{x}, i))$ relations, which can hinder readability. Moreover, the syntax and decision procedure of dFOATL have the limitation of quantifying over all possible values of variable and interval domains blindly, without taking the observed instances of operations in the trace into consideration.

To address these limitations, we propose a new syntactic construct that directly captures the notion of an operation in distributed protocols, enabling simultaneous quantification over all related variables and the interval in which the operation occurs, reflecting the way such actions are naturally observed in trace monitoring, and whose decision procedure performs quantification over known data, instead of inefficiently exploring all values of domains. This motivation leads to the iterative development of ACTL, presented in Chapter 5. Even though we introduce the idea of action, this logic is more restricted. While more cost-effective, it remains sufficiently constrained to express the cases that are the focus of our work.

Algorithm 4.1: The pseudocode of the predicate environment verification algorithm for formulas of dFOATL.

```
Function CHECK(ω, σ, φ):
    switch φ do
        case ¬φ′ do
            return not CHECK(ω, σ, φ′)
        case φ′ ∨ φ″ do
            return CHECK(ω, σ, φ′) or CHECK(ω, σ, φ″)
        case φ′ ∧ φ″ do
            return CHECK(ω, σ, φ′) and CHECK(ω, σ, φ″)
        case φ′ ⇒ φ″ do
            return not CHECK(ω, σ, φ′) or CHECK(ω, σ, φ″)
        case ∀ x . φ′ do
            foreach v ∈ D(x) do
                c ← CHECK(ω, σ[x ↦ v], φ′)
                if not c then return false
            return true
        case ∃ x . φ′ do
            foreach v ∈ D(x) do
                c ← CHECK(ω, σ[x ↦ v], φ′)
                if c then return true
            return false
        case ∀ i . φ′ do
            foreach (t_b, t_e) ∈ ℕ × ℕ do
                c ← CHECK(ω, σ[i ↦ (t_b, t_e)], φ′)
                if not c then return false
            return true
        case ∃ i . φ′ do
            foreach (t_b, t_e) ∈ ℕ × ℕ do
                c ← CHECK(ω, σ[i ↦ (t_b, t_e)], φ′)
                if c then return true
            return false
        case Holds(p(x̄), i) do
            (t_b, t_e) ← σ(i)
            foreach (t_b′, t_e′) ∈ ω(p, σ(x̄)) do
                if t_b′ ≤ t_b < t_e ≤ t_e′ then return true
            return false
        case Occurs(p(x̄), i) do
            (t_b, t_e) ← σ(i)
            foreach (t_b′, t_e′) ∈ ω(p, σ(x̄)) do
                if t_b ≤ t_e′ ∧ t_b′ ≤ t_e then return true
            return false
        case x_1 = x_2 do
            return σ(x_1) = σ(x_2)
        case r(i_1, i_2) do
            return [[r]](σ(i_1), σ(i_2))
```

5

Action Temporal Logic

Contents

In this chapter, we introduce Action Temporal Logic (ACTL), a restricted fragment of dFOATL that incorporates the notion of actions. We present its formal syntax and semantics in Section 5.1, define an encoding into dFOATL and prove its correctness in Section 5.2, illustrate its utility through the specification of implementation-independent properties of two distributed protocols in Section 5.3, and conclude with a decision procedure for ACTL, along with an analysis of its complexity and correctness in Section 5.4.

5.1 Syntax and Semantics

We introduce the definition of the syntax of ACTL formulas in Section 5.1.2, and subsequently formalize the semantics of the ACTL proposed logic fragment in Section 5.1.3. Before doing so, however, we provide a brief explanation of the notion of an action — the central construct from which the logic derives its name — in Section 5.1.1.

5.1.1 Actions

The addition of actions allows us to specify more effectively system properties by combining these operations with the relations inherited from dFOATL, while abstracting away from the underlying implementation details.

Actions represent system operations within a specific domain. They are characterized by their input parameters, output results, and the time interval during which the operation occurs. We assume all actions are total, meaning they terminate and yield complete results, abstracting away from intermediate states.

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 name of the action or operation, $x_1, \ldots, x_n$ are the input parameters, $y_1, \ldots, y_m$ are the output values produced by the action, and i is the time interval over which the action takes place.

This concept of actions was inspired by the formalization of DHTs proposed by Policarpo et al. [22].

5.1.2 Syntax

Let $\mathcal{V}$ be the set of finite domain variable symbols or constant values, $\mathcal{I}$ 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:

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

where $a \in \mathcal{A}$ is an action symbol, $x \in \mathcal{V}$ is a domain variable symbol or a constant, $\overline{x}$ and $\overline{y}$ are, respectively, finite sequences of domain variable symbols or constants $x_1, \ldots, x_{|x|}$ 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 formally in the same way as for dFOATL in Section 4.1.1.

Note that the differences between the temporal logic fragments of ACTL and dFOATL are highlighted using boxes in the respective syntax definitions of formulas: φ for ACTL, given above, and ϕ for dFOATL, presented in Section 4.1.1.

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:

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

5.1.3 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), as is the case with dFOATL:

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

When x is a constant, $\sigma(x) = x$.

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:

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

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

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.

The semantics of Allen's interval relations is defined formally in the same way as for dFOATL in Section 4.1.2.

For clarity and conciseness, we have adopted a slight abuse of notation by abbreviating point-wise assignments of values to domain variable symbols in the variable environment σ using sequences:

$$\sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, \ldots] \;\equiv\; \sigma[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, y_1 \mapsto w_1, \ldots, y_m \mapsto w_m, \ldots]$$

5.2 Translation from ACTL to dFOATL

This section presents a complete translation from the ACTL framework to the dFOATL framework. The translation of ACTL formulas φ into ϕ formulas of dFOATL is described in Section 5.2.1 via an encoding function, supported by derived dFOATL relations. Similarly, Section 5.2.2 introduces another encoding function that translates ACTL traces into dFOATL predicate environments. Finally, Section 5.2.3 presents an inductive proof establishing the equivalence of the two frameworks, supported by a lemma.

5.2.1 Translation of Formulas

Let F be the encoding function that, given a formula φ following the syntax of ACTL, produces a formula ϕ following the syntax of dFOATL. The translation of formulas is captured by this function as $F(\varphi) = \phi$:

$$F(\neg\varphi) = \neg F(\varphi)$$

$$F(\varphi_1 \vee \varphi_2) = F(\varphi_1) \vee F(\varphi_2)$$

$$F(\varphi_1 \wedge \varphi_2) = F(\varphi_1) \wedge F(\varphi_2)$$

$$F(\varphi_1 \Rightarrow \varphi_2) = F(\varphi_1) \Rightarrow F(\varphi_2)$$

$$F(\forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi) = \forall\, \overline{x}, \overline{y} \,.\, \forall\, i \,.\, \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \Rightarrow F(\varphi)$$

$$F(\exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi) = \exists\, \overline{x}, \overline{y} \,.\, \exists\, i \,.\, \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \Rightarrow F(\varphi)$$

$$F(x_1 = x_2) = (x_1 = x_2)$$

$$F(r(i_1, i_2)) = r(i_1, i_2)$$

where each action symbol a from $\mathcal{A}$ is mapped to a predicate symbol in $\mathcal{P}$, and $\mathit{HoldsMax}(p(\overline{x}), i)$ is an auxiliary relation defined in terms of other dFOATL constructs:

$$\mathit{HoldsMax}(p(\overline{x}), i) \iff \mathit{Holds}(p(\overline{x}), i) \wedge \neg\exists\, \hat{i} \,.\, \mathit{In}(i, \hat{i}) \wedge \mathit{Holds}(p(\overline{x}), \hat{i})$$

Informally, $\mathit{HoldsMax}(p(\overline{x}), i)$ is true when the predicate $p(\overline{x})$ holds maximally in the interval i, i.e., it holds throughout the entirety of the interval i, and there exists no strictly larger interval $\hat{i}$ such that $\mathit{In}(i, \hat{i})$ and $p(\overline{x})$ also holds throughout $\hat{i}$. In other words, i is a maximal interval for which $p(\overline{x})$ holds without being properly contained in any other such interval.

$\mathit{In}(i_1, i_2)$ is a derived auxiliary interval relation that holds when i_1 is strictly contained in i_2; put formally:

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

5.2.2 Translation of Traces

Let T be the encoding function that, given a trace τ as defined in Section 5.1.3, produces a predicate environment ω as defined in Section 4.1.2. The translation of traces is defined by this function as $T(\tau) = \omega$, where each begin-event and end-event symbol from $\mathcal{E}$ corresponding to the same occurrence of an action a in the trace τ are mapped to a new entry in ω. Each such entry records the action a and its input and output parameters and associates it with the pair of time points at which the begin-event and end-event occur in τ:

$$T(\tau)(a, \overline{v}, \overline{w}) \ni (t_b, t_e) \iff \mathbf{exists}\ k\ .\ \tau[t_b] \ni B\langle a\rangle_k(\overline{v})\ \mathbf{and}\ \tau[t_e] \ni E\langle a\rangle_k(\overline{w})$$

5.2.3 Equivalence of ACTL and dFOATL

Theorem (Equivalence of ACTL and dFOATL)**.** Given a variable environment σ, the verification of a formula φ over a trace τ is equivalent to the verification of $F(\varphi) = \phi$ with $T(\tau) = \omega$, which is denoted as:

$$\tau, \sigma \models_{\text{ACTL}} \varphi \iff T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi)$$

Proof (Equivalence of ACTL and dFOATL)**.** The proof of this theorem was inductively done on the structure of φ and is present in Figure 5.1 and Figure 5.2. It makes use of a lemma which is present in Figure 5.3.

5.3 Specification of Distributed Protocols

This section illustrates the expressive power of the ACTL framework through the specification of operations and correctness properties of representative distributed protocols. Leveraging the newly introduced notion of actions — which accept input parameters, yield outputs, occur over time intervals, and are subject to quantification — ACTL enables concise yet rigorous formalization of system behavior. The 2PC protocol is specified in Section 5.3.1, followed by a specification of a DHT protocol in Section 5.3.2. In these specifications, the wildcard ($-$) denotes a parameter that is not relevant within nested quantifiers or to the property itself.

5.3.1 Two-Phase Commit

The Two-Phase Commit (2PC) protocol was introduced in Section 2.3. We assume the same model with a single coordinator and multiple uniquely identified participant nodes. For more details regarding how

Figure 5.1: Part 1 of the inductive proof of the Equivalence of ACTL and dFOATL Theorem.

Base case: $\varphi = (x_1 = x_2)$

	$\tau, \sigma \models_{\text{ACTL}} \varphi$
→ by case $\varphi = (x_1 = x_2)$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} x_1 = x_2$
→ by ACTL semantics	$\Leftrightarrow \sigma(x_1) = \sigma(x_2)$
← by dFOATL semantics	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} x_1 = x_2$
← by def. of $F(\varphi)$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(x_1 = x_2)$
← by case $\varphi = (x_1 = x_2)$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi)$

Base case: $\varphi = r(i_1, i_2)$

	$\tau, \sigma \models_{\text{ACTL}} \varphi$
→ by case $\varphi = r(i_1, i_2)$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} r(i_1, i_2)$
→ by ACTL semantics	$\Leftrightarrow [\![r]\!](\sigma(i_1), \sigma(i_2))$
← by dFOATL semantics	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} r(i_1, i_2)$
← by def. of $F(\varphi)$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(r(i_1, i_2))$
← by case $\varphi = r(i_1, i_2)$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi)$

Inductive case: $\varphi = \neg\varphi'$

	$\tau, \sigma \models_{\text{ACTL}} \varphi$
→ by case $\varphi = \neg\varphi'$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \neg\varphi'$
→ by ACTL semantics	$\Leftrightarrow \mathbf{not}\ \tau, \sigma \models_{\text{ACTL}} \varphi'$
→ by induction hypothesis	$\Leftrightarrow \mathbf{not}\ T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi')$
← by dFOATL semantics	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} \neg F(\varphi')$
← by def. of $F(\varphi)$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\neg\varphi')$
← by case $\varphi = \neg\varphi'$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi)$

Inductive case: $\varphi = \varphi' \vee \varphi''$

	$\tau, \sigma \models_{\text{ACTL}} \varphi$
→ by case $\varphi = \varphi' \vee \varphi''$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi' \vee \varphi''$
→ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi'\ \mathbf{or}\ \tau, \sigma \models_{\text{ACTL}} \varphi''$
→ by induction hypothesis	$\Leftrightarrow T(\tau), \sigma \models_{\text{ACTL}} F(\varphi')\ \mathbf{or}\ T(\tau), \sigma \models_{\text{ACTL}} F(\varphi'')$
← by dFOATL semantics	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi') \vee F(\varphi'')$
← by def. of $F(\varphi)$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi' \vee \varphi'')$
← by case $\varphi = \varphi' \vee \varphi''$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi)$

Inductive case: $\varphi = \varphi' \wedge \varphi''$

	$\tau, \sigma \models_{\text{ACTL}} \varphi$
→ by case $\varphi = \varphi' \wedge \varphi''$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi' \wedge \varphi''$
→ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi'\ \mathbf{and}\ \tau, \sigma \models_{\text{ACTL}} \varphi''$
→ by induction hypothesis	$\Leftrightarrow T(\tau), \sigma \models_{\text{ACTL}} F(\varphi')\ \mathbf{and}\ T(\tau), \sigma \models_{\text{ACTL}} F(\varphi'')$
← by dFOATL semantics	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi') \wedge F(\varphi'')$
← by def. of $F(\varphi)$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi' \wedge \varphi'')$
← by case $\varphi = \varphi' \wedge \varphi''$	$\Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi)$

it works, the reader is referred back to that section.

This formalization reformulates the operations of the 2PC protocol using ACTL actions. They describe the full behavior previously presented and were written in the form of ACTL actions parameterized by nodes n, transactions t, data items d, and vote outcomes v, where v can take the constant values commit or abort.

Figure 5.2: Part 2 of the inductive proof of the Equivalence of ACTL and dFOATL Theorem.

Inductive case: $\varphi = \varphi' \Rightarrow \varphi''$

$$
\begin{array}{ll}
 & \tau, \sigma \models_{\text{ACTL}} \varphi \\
\rightarrow \text{ by case } \varphi = \varphi' \Rightarrow \varphi'' & \Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi' \Rightarrow \varphi'' \\
\rightarrow \text{ by ACTL semantics} & \Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi' \textbf{ implies } \tau, \sigma \models_{\text{ACTL}} \varphi'' \\
\rightarrow \text{ by induction hypothesis} & \Leftrightarrow T(\tau), \sigma \models_{\text{ACTL}} F(\varphi') \textbf{ implies } T(\tau), \sigma \models_{\text{ACTL}} F(\varphi'') \\
\leftarrow \text{ by dFOATL semantics} & \Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi') \Rightarrow F(\varphi'') \\
\leftarrow \text{ by def. of } F(\varphi) & \Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi' \Rightarrow \varphi'') \\
\leftarrow \text{ by case } \varphi = \varphi' \Rightarrow \varphi'' & \Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi)
\end{array}
$$

Inductive case: $\varphi = \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$

$$
\begin{array}{ll}
 & \tau, \sigma \models_{\text{ACTL}} \varphi \\
\rightarrow \text{ by case } \varphi = \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi' & \Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi' \\
\rightarrow \text{ by ACTL semantics} & \Leftrightarrow \textbf{forall } (t_b, t_e, \overline{v}, \overline{w}) \in \mathit{occurrences}(\tau, a) \,.\, \tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{ACTL}} \varphi' \\
\rightarrow \text{ by def. of } \mathit{occurrences}(\tau, a) & \Leftrightarrow \textbf{forall } t_b, t_e, \overline{v}, \overline{w} \,.\, \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}) \\
 & \qquad \textbf{implies } \tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{ACTL}} \varphi' \\
\rightarrow \text{ by induction hypothesis} & \Leftrightarrow \textbf{forall } t_b, t_e, \overline{v}, \overline{w} \,.\, \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}) \\
 & \qquad \textbf{implies } T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} F(\varphi') \\
\leftarrow \text{ by def. of } T(\tau) & \Leftrightarrow \textbf{forall } t_b, t_e, \overline{v}, \overline{w} \,.\, T(\tau)(a, \overline{v}, \overline{w}) \ni (t_b, t_e) \\
 & \qquad \textbf{implies } T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} F(\varphi') \\
\leftarrow \text{ by } \mathit{HoldsMax} \text{ lemma} & \Leftrightarrow \textbf{forall } t_b, t_e, \overline{v}, \overline{w} \,.\, T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \\
 & \qquad \textbf{implies } T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} F(\varphi') \\
\leftarrow \text{ by dFOATL semantics } (\Rightarrow) & \Leftrightarrow \textbf{forall } t_b, t_e, \overline{v}, \overline{w} \,.\, T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \Rightarrow F(\varphi') \\
\leftarrow \text{ by dFOATL semantics } (\forall) & \Leftrightarrow \textbf{forall } \overline{v}, \overline{w} \,.\, T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}] \models_{\text{dFOATL}} \forall i \,.\, \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \Rightarrow F(\varphi') \\
\leftarrow \text{ by dFOATL semantics } (\forall) & \Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} \forall \overline{x}, \overline{y} \,.\, \forall i \,.\, \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \Rightarrow F(\varphi') \\
\leftarrow \text{ by def. of } F(\varphi) & \Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi') \\
\leftarrow \text{ by case } \varphi = \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi' & \Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi)
\end{array}
$$

Inductive case: $\varphi = \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$

$$
\begin{array}{ll}
 & \tau, \sigma \models_{\text{ACTL}} \varphi \\
\rightarrow \text{ by case } \varphi = \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi' & \Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi' \\
\rightarrow \text{ by ACTL semantics} & \Leftrightarrow \textbf{exists } (t_b, t_e, \overline{v}, \overline{w}) \in \mathit{occurrences}(\tau, a) \,.\, \tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{ACTL}} \varphi' \\
\rightarrow \text{ by def. of } \mathit{occurrences}(\tau, a) & \Leftrightarrow \textbf{exists } t_b, t_e, \overline{v}, \overline{w} \,.\, \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}) \\
 & \qquad \textbf{implies } \tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{ACTL}} \varphi' \\
\rightarrow \text{ by induction hypothesis} & \Leftrightarrow \textbf{exists } t_b, t_e, \overline{v}, \overline{w} \,.\, \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}) \\
 & \qquad \textbf{implies } T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} F(\varphi') \\
\leftarrow \text{ by def. of } T(\tau) & \Leftrightarrow \textbf{exists } t_b, t_e, \overline{v}, \overline{w} \,.\, T(\tau)(a, \overline{v}, \overline{w}) \ni (t_b, t_e) \\
 & \qquad \textbf{implies } T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} F(\varphi') \\
\leftarrow \text{ by } \mathit{HoldsMax} \text{ lemma} & \Leftrightarrow \textbf{exists } t_b, t_e, \overline{v}, \overline{w} \,.\, T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \\
 & \qquad \textbf{implies } T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} F(\varphi') \\
\leftarrow \text{ by dFOATL semantics } (\Rightarrow) & \Leftrightarrow \textbf{exists } t_b, t_e, \overline{v}, \overline{w} \,.\, T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{dFOATL}} \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \Rightarrow F(\varphi') \\
\leftarrow \text{ by dFOATL semantics } (\exists) & \Leftrightarrow \textbf{exists } \overline{v}, \overline{w} \,.\, T(\tau), \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}] \models_{\text{dFOATL}} \exists i \,.\, \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \Rightarrow F(\varphi') \\
\leftarrow \text{ by dFOATL semantics } (\exists) & \Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} \exists \overline{x}, \overline{y} \,.\, \exists i \,.\, \mathit{HoldsMax}(a(\overline{x}, \overline{y}), i) \Rightarrow F(\varphi') \\
\leftarrow \text{ by def. of } F(\varphi) & \Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi') \\
\leftarrow \text{ by case } \varphi = \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi' & \Leftrightarrow T(\tau), \sigma \models_{\text{dFOATL}} F(\varphi)
\end{array}
$$

- $\textbf{request}(n, t, d) \overset{i}{\rightsquigarrow} (v)$: during interval i, the coordinator requests node n to prepare for transaction t with data d; the operation completes when the node sends its response with its vote to either commit or abort that transaction.

Figure 5.3: The $HoldsMax$ lemma, used in the Equivalence of ACTL and dFOATL Proof.

Lemma: $HoldsMax$ where $\sigma' = \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)]$

$$
\begin{aligned}
& T(\tau), \sigma' \models_{\text{dFOATL}} HoldsMax(a(\overline{x}, \overline{y}), i) \\
\Leftrightarrow\ & T(\tau), \sigma' \models_{\text{dFOATL}} Holds(a(\overline{x}, \overline{y}), i) \wedge \neg \exists \hat{i} \,.\, In(i, \hat{i}) \wedge Holds(a(\overline{x}, \overline{y}), \hat{i}) \\
\Leftrightarrow\ & T(\tau), \sigma' \models_{\text{dFOATL}} Holds(a(\overline{x}, \overline{y}), i) \textbf{ and } T(\tau), \sigma' \models_{\text{dFOATL}} \neg \exists \hat{i}. \, In(i, \hat{i}) \wedge Holds(a(\overline{x}, \overline{y}), \hat{i}) \\
\Leftrightarrow\ & T(\tau), \sigma' \models_{\text{dFOATL}} Holds(a(\overline{x}, \overline{y}), i) \textbf{ and not } T(\tau), \sigma' \models_{\text{dFOATL}} \exists \hat{i}. \, In(i, \hat{i}) \wedge Holds(a(\overline{x}, \overline{y}), \hat{i}) \\
\Leftrightarrow\ & T(\tau), \sigma' \models_{\text{dFOATL}} Holds(a(\overline{x}, \overline{y}), i) \textbf{ and not exists } \hat{t_b}, \hat{t_e} \,.\, T(\tau), \sigma'[\hat{i} \mapsto (\hat{t_b}, \hat{t_e})] \models_{\text{dFOATL}} In(i, \hat{i}) \wedge Holds(a(\overline{x}, \overline{y}), \hat{i}) \\
\Leftrightarrow\ & T(\tau), \sigma' \models_{\text{dFOATL}} Holds(a(\overline{x}, \overline{y}), i) \\
& \qquad \textbf{and not exists } \hat{t_b}, \hat{t_e} \,.\, T(\tau), \sigma'[\hat{i} \mapsto (\hat{t_b}, \hat{t_e})] \models_{\text{dFOATL}} In(i, \hat{i}) \\
& \qquad\qquad \textbf{and } T(\tau), \sigma'[\hat{i} \mapsto (\hat{t_b}, \hat{t_e})] \models_{\text{dFOATL}} Holds(a(\overline{x}, \overline{y}), \hat{i}) \\
\Leftrightarrow\ & \textbf{exists } t_{b'}, t_{e'} \,.\, \{ T(\tau)(a, \overline{v}, \overline{w}) \ni (t_{b'}, t_{e'}) \wedge t_{b'} \leq t_b < t_e \leq t_{e'} \} \\
& \qquad \textbf{and not exists } \hat{t_b}, \hat{t_e} \,.\, \{ T(\tau), \sigma'[\hat{i} \mapsto (\hat{t_b}, \hat{t_e})] \models_{\text{dFOATL}} In(i, \hat{i}) \\
& \qquad\qquad \textbf{and exists } \hat{t_{b'}}, \hat{t_{e'}} \,.\, T(\tau)(a, \overline{v}, \overline{w}) \ni (\hat{t_{b'}}, \hat{t_{e'}}) \wedge t_{b'} \leq \hat{t_b} < \hat{t_e} \leq \hat{t_{e'}} \} \\
\Leftrightarrow\ & \textbf{exists } t_{b'}, t_{e'} \,.\, \{ T(\tau)(a, \overline{v}, \overline{w}) \ni (t_{b'}, t_{e'}) \wedge t_{b'} \leq t_b < t_e \leq t_{e'} \} \\
& \qquad \textbf{and not exists } \hat{t_b}, \hat{t_e} \,.\, \{ T(\tau), \sigma'[\hat{i} \mapsto (\hat{t_b}, \hat{t_e})] \models_{\text{dFOATL}} Starts(i, \hat{i}) \vee During(i, \hat{i}) \vee Finishes(i, \hat{i}) \\
& \qquad\qquad \textbf{and exists } \hat{t_{b'}}, \hat{t_{e'}} \,.\, T(\tau)(a, \overline{v}, \overline{w}) \ni (\hat{t_{b'}}, \hat{t_{e'}}) \wedge t_{b'} \leq \hat{t_b} < \hat{t_e} \leq \hat{t_{e'}} \} \\
\Leftrightarrow\ & T(\tau)(a, \overline{v}, \overline{w}) \ni (t_b, t_e)
\end{aligned}
$$

- $\mathbf{instruct}(v, t) \overset{i}{\rightsquigarrow} (d)$: during interval i, the coordinator instructs all participant nodes to commit or abort transaction t; the operation completes upon receipt of acknowledgment data d confirming the execution of the instruction.

Given the nature of ACTL actions, which take inputs and produce outputs, some of the previously defined operations were merged and simplified, as follows in Table 5.1.

Table 5.1: Simplification iterative process in the definition of 2PC operations.

FOLTL	dFOATL	ACTL
$\mathbf{request}(n, t, d)$	$\mathbf{request}(n, t, d)$	$\mathbf{request}(n, t, d) \overset{i}{\rightsquigarrow} (v)$
$\mathbf{v\text{-}commit}(n, t)$	$\mathbf{vote}(n, v, t)$	
$\mathbf{v\text{-}abort}(n, t)$		
$\mathbf{i\text{-}commit}(t)$	$\mathbf{instruct}(v, t)$	$\mathbf{instruct}(v, t) \overset{i}{\rightsquigarrow} (d)$
$\mathbf{i\text{-}abort}(t)$		
$\mathbf{e\text{-}commit}(n, t)$	$\mathbf{execute}(n, v, t)$	
$\mathbf{e\text{-}abort}(n, t)$		

The same example correctness properties from Section 2.3 are reformulated as ACTL formulas, providing an informative basis for comparison between the two representations.

(LIVENESS) **Eventual decision**: if the coordinator requests any participant node n to prepare for any transaction t, all participant nodes will eventually be instructed to either commit or abort that transaction.

$$\forall\, \mathbf{request}(n, t, -) \overset{i_1}{\rightsquigarrow} (-) \,.\, \exists\, \mathbf{instruct}(-, t) \overset{i_2}{\rightsquigarrow} (-) \,.\, Before(i_1, i_2)$$

(SAFETY) **Atomic commitment**: if the coordinator instructs all participant nodes to commit any transaction

t, then all nodes n must have voted to commit that transaction; if it instructs to abort t, then at least one node n must have voted to abort it.

$$\forall\, \mathbf{instruct}(v_1, t) \overset{i_1}{\rightsquigarrow} (-)\,.\,\big(v_1 = \text{commit} \Rightarrow \forall\, \mathbf{request}(n, t, -) \overset{i_2}{\rightsquigarrow} (v_2)\,.\,\mathit{Before}(i_2, i_1) \wedge v_1 = v_2\big)$$
$$\wedge\, \big(v_1 = \text{abort} \Rightarrow \exists\, \mathbf{request}(n, t, -) \overset{i_2}{\rightsquigarrow} (v_1)\,.\,\mathit{Before}(i_2, i_1)\big)$$

(SAFETY) **Commit justification**: if any node n executes a commit of any transaction t, then the coordinator must have instructed all participant nodes to commit that transaction, which must have been preceded by a vote to commit from that node and, in turn, a prepare request.

$$\forall\, \mathbf{instruct}(\text{commit}, t) \overset{i_1}{\rightsquigarrow} (-)\,.\,\forall\, \mathbf{request}(n, t, -) \overset{i_2}{\rightsquigarrow} (\text{commit})\,.\,\mathit{Before}(i_2, i_1)$$

Although the atomic commitment and commit justification properties are theoretically distinct and address different aspects of the protocol, we observe that, due to our merging of operations, the atomic commitment property as expressed through ACTL actions effectively subsumes the commit justification property when translated and written in ACTL syntax.

5.3.2 Distributed Hash Table

The definition of a generic Distributed Hash Table (DHT) protocol was introduced in Section 2.4. For more details regarding how it works, the reader is referred back to that section, where we also specified correctness properties using FOLTL. While this logic is well-suited for reasoning over temporal traces, the new action-based formalism makes ACTL a more appropriate choice for specifying distributed protocol behavior.

This formalization reformulates the operations of the DHT protocol using ACTL actions, and then specifies correctness value, key and structural properties using ACTL formulas.

Using ACTL's interval relations, derived from dFOATL and ultimately from ATL, we can define auxiliary predicates that improve the readability of our formulas. The $\mathit{In}(i_1, i_2)$ and $\mathit{Intersects}(i_1, i_2)$ predicates are particularly useful for specifying DHT properties. The $\mathit{In}(i_1, i_2)$ predicate was previously introduced in Section 5.2.1. The $\mathit{Intersects}(i_1, i_2)$ predicate holds if i_1 and i_2 are ongoing during one common instant; put formally:

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

Some DHT properties hold only under specific conditions. For example, if two lookups for the same key occur while the network is stable and no store operation is in progress, they should return the same value. We distinguish two kinds of such conditions: *runtime regimens*, describing which operations are

currently executing, and *network states*, describing the system's current configuration.

Assuming a network of uniquely identified nodes that store key–value pairs, we model functional, safety, protocol-independent properties using ACTL, where DHT operations are represented as ACTL actions parameterized by nodes n, keys k, and values v. Additionally, predicates can be modeled as actions with input but no output parameters, while network states and regimens are modeled as actions without input or output, using only their associated time intervals.

- $\mathbf{store}(n, k, v) \overset{i}{\rightsquigarrow} (n')$: 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.
- $\mathbf{lookup}(n, k) \overset{i}{\rightsquigarrow} (n', v)$: 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.
- $\mathbf{findnode}(n, k) \overset{i}{\rightsquigarrow} (n', n'')$: 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.
- $\mathbf{join}(n)\ ^{i}$: during interval i, node n joins the network, resulting in a configuration in which node n is a member of the network.
- $\mathbf{leave}(n)\ ^{i}$: 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.
- $\mathbf{fail}(n)\ ^{i}$: during interval i, node n fails without performing maintenance operations, resulting in a new configuration where n is no longer a member of the network.
- $\mathbf{responsible}(n, k)\ ^{i}$: during interval i, node n is responsible for key k.
- $\mathbf{member}(n)\ ^{i}$: during interval i, node n is a member of the network.
- $\mathbf{ideal}\ ^{i}$: 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).
- $\mathbf{readonly}\ ^{i}$: 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).
- $\mathbf{stable}\ ^{i}$: 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.

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

(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.

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

(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.

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

(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.

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

(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.

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

5.4 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. The pseudocode of the procedure is provided in Section 5.4.1, together with explanations and an auxiliary trace preprocessing function designed to improve efficiency and reduce overall complexity — this aspect is further discussed in Section 5.4.2. Finally, Section 5.4.3 presents a proof establishing the correctness of the proposed decision procedure with respect to the formal semantics of ACTL present in Section 5.1.3.

5.4.1 Pseudocode

The pseudocode in Algorithm 5.1 defines the CHECK(τ, σ, φ) function, 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 $(t_b, t_e, \overline{v}, \overline{w})$ corre-

Algorithm 5.1: The pseudocode of the trace verification algorithm for formulas of ACTL.

```
Function CHECK(τ, σ, φ):
    switch φ do
        case ¬φ′ do
            return not CHECK(τ, σ, φ′)
        case φ′ ∨ φ″ do
            return CHECK(τ, σ, φ′) or CHECK(τ, σ, φ″)
        case φ′ ∧ φ″ do
            return CHECK(τ, σ, φ′) and CHECK(τ, σ, φ″)
        case φ′ ⇒ φ″ do
            return not CHECK(τ, σ, φ′) or CHECK(τ, σ, φ″)
        case ∀a(x̄) ⇝^i (ȳ) . φ′ do
            foreach (t_b, t_e, v̄, w̄) ∈ M_τ[a] do
                c ← CHECK(τ, σ[x̄ ↦ v̄, ȳ ↦ w̄, i ↦ (t_b, t_e)], φ′)
                if not c then return false
            return true
        case ∃a(x̄) ⇝^i (ȳ) . φ′ do
            foreach (t_b, t_e, v̄, w̄) ∈ M_τ[a] do
                c ← CHECK(τ, σ[x̄ ↦ v̄, ȳ ↦ w̄, i ↦ (t_b, t_e)], φ′)
                if c then return true
            return false
        case x_1 = x_2 do
            return σ(x_1) = σ(x_2)
        case r(i_1, i_2) do
            return [[r]](σ(i_1), σ(i_2))
```

sponding 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 5.1.3: for each action $a \in \mathcal{A}$, the set $\mathcal{M}_\tau[a]$ corresponds to $occurrences(\tau, a)$, 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 CHECK(τ, σ, φ), though it is implicitly carried with τ through recursive calls.

The pseudocode in Algorithm 5.2 defines the PREPROCESSTRACE(τ) function, which constructs the mapping $\mathcal{M}_\tau$ from the trace τ.

Algorithm 5.2: The pseudocode of the preprocessing algorithm for traces of ACTL.

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

5.4.2 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 $\mathcal{O}(e \cdot n)$, 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 $\mathcal{O}(n)$. 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., $\mathcal{O}(1)$.

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

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, $\mathcal{O}(1)$, using the precomputed mapping $\mathcal{M}_\tau$. For each occurrence found, the algorithm recursively checks a subformula. Therefore, the overall worst-case complexity is $\mathcal{O}(o^q)$, 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 $|\tau|$.

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

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 5.3) 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:

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

Even though the worst-case complexity of the decision procedure of ACTL belongs to the same class as that of dFOATL formulas presented in Section 4.3, in practice this procedure performs significantly better. Here, the base of the exponentiation is bounded by the maximum number of occurrences of any operation (o), whereas in the dFOATL case it depends on the total number of possible values in the largest variable domain (d). Moreover, the effective exponent, q, is typically smaller here, since it corresponds to the number of actions in the formula, while in dFOATL each quantification must be handled individually, leading to a higher bound.

Besides being more efficient than dFOATL in terms of the decision procedure, this new formalism also enables clearer and more expressive specifications. While dFOATL addressed the need to represent distributed protocol operations that span intervals of time, it did not capture their inherent input–output separation. In contrast, the current logic not only improves quantification but also offers a syntax more naturally tailored to distributed protocol behavior. Moreover, ACTL may allow for simpler proofs of correctness.

5.4.3 Procedure Correctness

This section establishes the correctness of the proposed decision procedure for ACTL, formally proving that the algorithm faithfully implements the semantics defined in Section 5.1.3.

Theorem (Correctness of the Decision Procedure for ACTL)**.** The function CHECK(τ, σ, φ) correctly determines whether a formula φ holds over a trace τ under the variable environment σ, according to the formal semantics of ACTL:

$$\text{CHECK}(\tau, \sigma, \varphi) = true \iff \tau, \sigma \models_{\text{ACTL}} \varphi$$

Proof (Correctness of the Decision Procedure for ACTL)**.** The proof of this theorem was inductively done on the structure of φ and is present in Figure 5.4.

Figure 5.4: The inductive proof of the Correctness of the Decision Procedure for ACTL Theorem.

Base case $\varphi = (x_1 = x_2)$

	$\textsc{Check}(\tau, \sigma, \varphi)$
$\rightarrow$ by case $\varphi = (x_1 = x_2)$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, x_1 = x_2)$
$\rightarrow$ by def. of $\textsc{Check}(\tau, \sigma, \varphi)$	$\Leftrightarrow \sigma(x_1) = \sigma(x_2)$
$\leftarrow$ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} x_1 = x_2$
$\leftarrow$ by case $\varphi = (x_1 = x_2)$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi$

Base case $\varphi = r(i_1, i_2)$

	$\textsc{Check}(\tau, \sigma, \varphi)$
$\rightarrow$ by case $\varphi = r(i_1, i_2)$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, r(i_1, i_2))$
$\rightarrow$ by def. of $\textsc{Check}(\tau, \sigma, \varphi)$	$\Leftrightarrow [\![r]\!](\sigma(i_1), \sigma(i_2))$
$\leftarrow$ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} r(i_1, i_2)$
$\leftarrow$ by case $\varphi = r(i_1, i_2)$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi$

Inductive case: $\varphi = \neg\varphi'$

	$\textsc{Check}(\tau, \sigma, \varphi)$
$\rightarrow$ by case $\varphi = \neg\varphi'$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, \neg\varphi')$
$\rightarrow$ by def. of $\textsc{Check}(\tau, \sigma, \varphi)$	$\Leftrightarrow \textbf{not } \textsc{Check}(\tau, \sigma, \varphi')$
$\rightarrow$ by induction hypothesis	$\Leftrightarrow \textbf{not } \tau, \sigma \models_{\text{ACTL}} \varphi'$
$\leftarrow$ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \neg\varphi'$
$\leftarrow$ by case $\varphi = \neg\varphi'$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi$

Inductive case: $\varphi = \varphi' \vee \varphi''$

	$\textsc{Check}(\tau, \sigma, \varphi)$
$\rightarrow$ by case $\varphi = \varphi' \vee \varphi''$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, \varphi' \vee \varphi'')$
$\rightarrow$ by def. of $\textsc{Check}(\tau, \sigma, \varphi)$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, \varphi') \textbf{ or } \textsc{Check}(\tau, \sigma, \varphi'')$
$\rightarrow$ by induction hypothesis	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi' \textbf{ or } \tau, \sigma \models_{\text{ACTL}} \varphi''$
$\leftarrow$ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi' \vee \varphi''$
$\leftarrow$ by case $\varphi = \varphi' \vee \varphi''$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi$

Inductive case: $\varphi = \varphi' \wedge \varphi''$

	$\textsc{Check}(\tau, \sigma, \varphi)$
$\rightarrow$ by case $\varphi = \varphi' \wedge \varphi''$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, \varphi' \wedge \varphi'')$
$\rightarrow$ by def. of $\textsc{Check}(\tau, \sigma, \varphi)$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, \varphi') \textbf{ and } \textsc{Check}(\tau, \sigma, \varphi'')$
$\rightarrow$ by induction hypothesis	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi' \textbf{ and } \tau, \sigma \models_{\text{ACTL}} \varphi''$
$\leftarrow$ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi' \wedge \varphi''$
$\leftarrow$ by case $\varphi = \varphi' \wedge \varphi''$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi$

Inductive case: $\varphi = \varphi' \Rightarrow \varphi''$

	$\textsc{Check}(\tau, \sigma, \varphi)$
$\rightarrow$ by case $\varphi = \varphi' \Rightarrow \varphi''$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, \varphi' \Rightarrow \varphi'')$
$\rightarrow$ by def. of $\textsc{Check}(\tau, \sigma, \varphi)$	$\Leftrightarrow \textbf{not } \textsc{Check}(\tau, \sigma, \varphi') \textbf{ or } \textsc{Check}(\tau, \sigma, \varphi'')$
$\rightarrow$ by induction hypothesis	$\Leftrightarrow \textbf{not } \tau, \sigma \models_{\text{ACTL}} \varphi' \textbf{ or } \tau, \sigma \models_{\text{ACTL}} \varphi''$
$\leftarrow$ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi' \Rightarrow \varphi''$
$\leftarrow$ by case $\varphi = \varphi' \Rightarrow \varphi''$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi$

Inductive case: $\varphi = \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$

	$\textsc{Check}(\tau, \sigma, \varphi)$
$\rightarrow$ by case $\varphi = \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi')$
$\rightarrow$ by def. of $\textsc{Check}(\tau, \sigma, \varphi)$	$\Leftrightarrow \textbf{forall } (t_b, t_e, \overline{v}, \overline{w}) \in \mathit{occurrences}(\tau, a) \,.\, \textsc{Check}(\tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)], \varphi')$
$\rightarrow$ by induction hypothesis	$\Leftrightarrow \textbf{forall } (t_b, t_e, \overline{v}, \overline{w}) \in \mathit{occurrences}(\tau, a) \,.\, \tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{ACTL}} \varphi'$
$\leftarrow$ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$
$\leftarrow$ by case $\varphi = \forall a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi$

Inductive case: $\varphi = \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$

	$\textsc{Check}(\tau, \sigma, \varphi)$
$\rightarrow$ by case $\varphi = \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$	$\Leftrightarrow \textsc{Check}(\tau, \sigma, \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi')$
$\rightarrow$ by def. of $\textsc{Check}(\tau, \sigma, \varphi)$	$\Leftrightarrow \textbf{exists } (t_b, t_e, \overline{v}, \overline{w}) \in \mathit{occurrences}(\tau, a) \,.\, \textsc{Check}(\tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)], \varphi')$
$\rightarrow$ by induction hypothesis	$\Leftrightarrow \textbf{exists } (t_b, t_e, \overline{v}, \overline{w}) \in \mathit{occurrences}(\tau, a) \,.\, \tau, \sigma[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{w}, i \mapsto (t_b, t_e)] \models_{\text{ACTL}} \varphi'$
$\leftarrow$ by ACTL semantics	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$
$\leftarrow$ by case $\varphi = \exists a(\overline{x}) \overset{i}{\rightsquigarrow} (\overline{y}) \,.\, \varphi'$	$\Leftrightarrow \tau, \sigma \models_{\text{ACTL}} \varphi$

6

Implementation

Contents

This chapter presents the implementation of our tool, ACTLChecker, described in detail in Section 6.3, including its main modules, components, and their respective responsibilities in the monitoring process.

Before reasoning about traces and ACTL formulas, it is necessary to obtain execution logs by instrumenting distributed protocol implementations, as discussed in Section 6.2. Additionally, we define a machine-readable specification language for the ACTL syntax to support automated monitoring and verification, presented in Section 6.1.

6.1 Specification Language for ACTL

We aim to develop an offline testing tool for ACTL that takes as input formulas specifying properties that must hold for a distributed protocol. Previously, we expressed a few properties in Section 5.3 using ACTL formulas following the syntax defined in Section 5.1.2. While these formulas are concise and readable for humans, they are not directly practical for machine interpretation.

To address this, we have designed a formal specification language that can be parsed by our monitoring tool using only alphanumeric characters readily available on standard keyboards. The language uses a prefix notation with nested parentheses, which preserves readability while enabling machine traversal and evaluation. Once familiar with both representations, the translation between mathematical formulas and the specification language is straightforward.

As an example, consider the lookup consistency and key consistency properties for DHT protocols, previously expressed in Section 5.3.2. In Listing 6.1, and Listing 6.2 we illustrate these same properties using our formal specification 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.

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

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

Listing 6.1: Lookup consistency property in the ACTL specification language.

(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.

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

```
(forall findnode f1 (- k) (- n1)
    (forall findnode f2 (- k) (- n2)
        (forall ideal i () ()
            (forall stable s () ()
                (implies
                    (and
                        (or (in f1 i) (equals f1 i))
                        (or (in f2 i) (equals f2 i))
                        (or (in f1 s) (equals f1 s))
                        (or (in f2 s) (equals f2 s))
                    )
                    (n1 = n2)
)   )   )   )   )
```

Listing 6.2: Key consistency property in the ACTL specification language.

6.2 Instrumentation of a DHT implementation

In this work, we distinguish between logs and traces. Traces are processed and structured execution records ready for formal reasoning or analysis, while logs are raw, unprocessed data collected directly from the system. This distinction is introduced purely for clarity within this work to maintain a consistent terminology.

To obtain representative execution traces from distributed systems, these systems must first be instrumented, i.e. extended with mechanisms that record relevant runtime information about the operations being executed. Instrumentation in distributed systems is challenging because multiple independent nodes operate concurrently, making it necessary to correctly correlate and order events despite message delays, clock skew, or node failures. A fuzzer is typically employed before this step to automatically generate diverse input scenarios and result in different execution paths, ensuring that the collected logs capture a broad range of system behaviors. Although developing such instrumentation lies beyond the scope of this work, it is a crucial and technically demanding task whose quality directly determines the reliability of any analysis performed on the resulting logs.

Fortunately, Policarpo et al. have already instrumented an implementation of the OpenChord DHT protocol [28] as part of ongoing work extending their previous contribution on specifying distributed protocols using interval temporal logics [22]. We therefore leverage their instrumented logs for our study.

An excerpt of a log in their format is shown in Listing 6.3. This format corresponds to the direct output of their instrumentation process. Their logs may also be enriched with additional information, such as maximal intervals representing node membership periods, inferred network states, and run-

time regimens, obtained by analyzing which operations are active at each instant. This supplementary information, while designed to support their analyses, is also valuable for our purposes.

Each log entry follows the following format:

```
<timestamp>, <operation>, <operation ID>, <fields*>
```

where:

- `<timestamp>` is the time the event occurred;
- `<operation>` is the type of operation (e.g., `Join`, `ReplyFindNode`). Reply messages are prefixed with `Reply` to distinguish them from initiating operations;
- `<operation ID>` is a unique identifier for the operation;
- `<fields*>` are fields relevant to the operation.

```
2025-03-23 17:34:26.649, EndStable
2025-03-23 17:34:26.650, Join, a900fe36, 488D
2025-03-23 17:34:27.016, FindNode, be67d936, 7BB3, 488D
2025-03-23 17:34:27.017, ReplyFindNode, be67d936, 7BB3, 7BB3
2025-03-23 17:34:27.397, ReplyJoin, a900fe36
2025-03-23 17:34:27.398, Member, 488D
2025-03-23 17:34:27.398, Stable
2025-03-23 17:36:30.113, EndReadOnly
2025-03-23 17:36:30.114, Store, f95852e0, 7BB3, 8C1E, 44E6
2025-03-23 17:36:30.163, ReplyStore, f95852e0, ADC0
2025-03-23 17:36:30.164, ReadOnly
```

Listing 6.3: Extract of a log got from the OpenChord instrumented implementation.

In this example, the network leaves a stable regimen when node `488D` joins. After the join completes, `488D` becomes a member, and the network returns to stability. Later, it exits a *read only* regimen as node `7BB3` performs a store operation for key–value pair (`8C1E`, `44E6`), re-entering the *read only* state once the operation finishes.

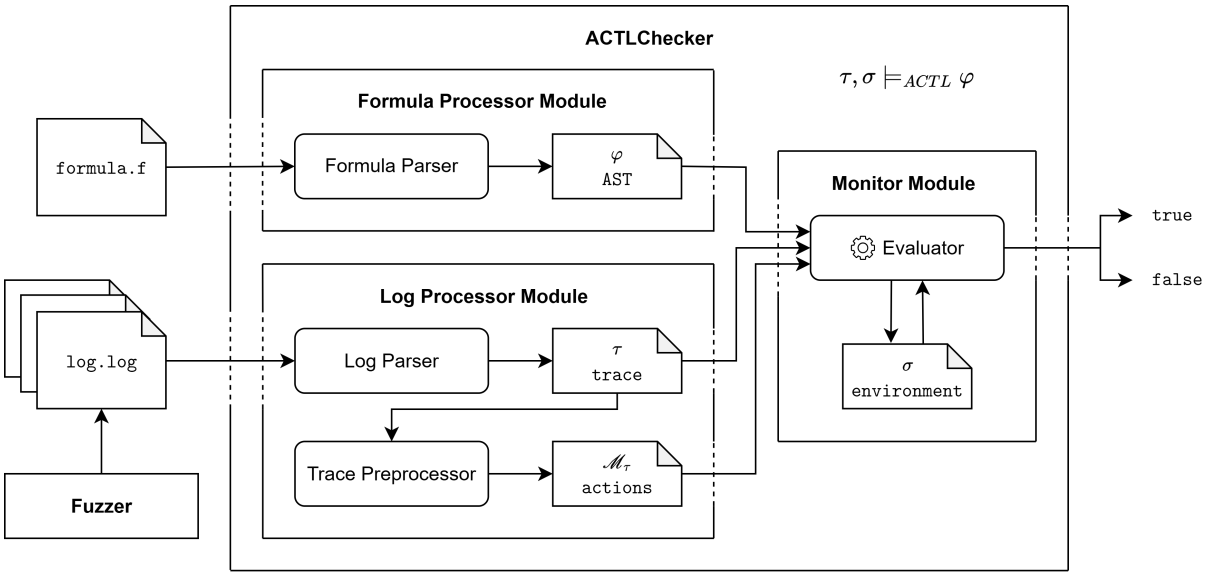
However, as established in our definition of a trace τ in Section 5.1.3 — which was also used in the design of the ACTL decision procedure presented in Section 5.4 — a trace is a sequence of sets of events different from this log representation. Since this τ definition underlies the algorithm we intend to implement in our monitoring tool for distributed protocols, it is necessary to transform the instrumented logs into the corresponding trace format compatible with our framework in order to conduct an evaluation on these OpenChord logs. As one can see, this can be done by traversing the log and adapting to its syntax (e.g. `Reply` corresponds to an end-event).

6.3 ACTLChecker

We have now established all the necessary steps that must precede the task of validation of distributed protocol implementations using an offline testing pipeline. The remaining step is the development of the tool that realizes this pipeline. This section presents the design and structure of that checker.

Because the tool operates over finite execution logs, its verification capabilities are inherently limited to safety properties. An overview of ACTLChecker is provided in Figure 6.1, which illustrates the general architecture of the system.

Figure 6.1: The general architecture of ACTLChecker.



To apply the tool, users first specify the correctness properties of the distributed protocol under analysis using the formal specification language introduced in Section 6.1. This language corresponds directly to the ACTL syntax, expressed in a machine-interpretable format. Subsequently, an implementation of the protocol must be instrumented and executed to collect logs of its behavior, as described in Section 6.2. A fuzzer is employed to generate a wide variety of execution scenarios, enabling the exploration of different system behaviors and the capture of representative traces.

Once the logs are collected, they may optionally undergo preprocessing to include additional information necessary for verifying particular properties or to conform to the input format required by ACTLChecker, present in Section 6.2. Finally, the tool analyzes the execution traces against the formal specifications, determining whether the recorded behaviors satisfy the specified properties.

ACTLChecker is implemented in Python and is openly accessible [1]. Each of the processor modules can be invoked individually, but running the monitor module is as straightforward as shown in Listing 6.4:

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

```
$ python checker.py -f formula.f -l log.log [-d] [-n]
```

Listing 6.4: Command to run in order to use ACTLChecker to monitor a log using a formula.

where both the formula and the log can be specified as file paths or strings. The `-d` flag enables debugging, while the `-n` flag limits the number of log lines to be considered. And the outcome, with the debugging option disabled, should appear like Listing 6.5 in just a matter of seconds:

```
Evaluation: true / false
```

Listing 6.5: Example outputs of ACTLChecker evaluation: `true` and `false`.

ACTLChecker is particularly suited for testing executions of complex, data-intensive systems characterized by interdependent operations and multiple correctness properties, such as distributed protocols. Its modules are all independent of the implementation of the protocol under testing, requiring only that the formulas and the generated logs be in the formats described previously. Below, we describe the main modules of the pipeline, namely the Formula Processor in Section 6.3.1, the Log Processor in Section 6.3.2, and the Monitor in Section 6.3.3.

6.3.1 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 6.1.

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 5.1.2.

6.3.2 Log Processor Module

The Log Processor module takes as input an execution log generated during instrumentation, as described in Section 6.2. 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 5.1.3. 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 5.4.1, thereby reducing complexity during trace evaluation.

In practice, these two components have been optimized and merged so that the trace and mapping are created simultaneously, improving the efficiency of the log processing stage.

6.3.3 Monitor Module

The Monitor module is the core of ACTLChecker and the "brain" 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 5.4, which closely follows the semantics defined in Section 5.1.3. During evaluation, the environment σ, implemented as a Python list, 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.

7

Evaluation

Contents

This chapter evaluates the behavior of ACTLChecker, our new tool that is described alongside the necessary steps to prepare the required input files for its execution in Chapter 6. The evaluation of our implementation focuses on the same criteria that originally motivated its development: achieving efficient testing (beyond mere 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 [23] as a case study, whose essential concepts were introduced in Section 2.4. This choice is justified by two main factors:

- The availability of pre-collected logs from instrumented executions of an existing DHT implementation — OpenChord [28] — obtained in the work of Policarpo et al. [22], 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 5.3.2. Naturally, these specifications are provided to the tool in the formal language defined in Section 6.1.

In this chapter, we evaluate whether ACTLChecker is capable of correctly verifying or falsifying properties specified in ACTL against traces of real distributed systems, namely DHTs. This question is addressed in Section 7.1.

7.1 Functionality of ACTLChecker

Although we are confident that any distributed protocol can be specified in ACTL given a prior axiomatization of its operations, using DHTs as an example, 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 correctness specifications of the properties described in Section 5.3.2, 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 7.1 below summarizes the results of this monitoring experiment across the eight properties:

Table 7.1: Properties, their classes, and whether ACTLChecker can validate them against execution logs of OpenChord.

Class	Property	ACTLChecker
VALUE	Lookup consistency	`true`
	Value consistency	`true`
	Value freshness	`true`
KEY	Key consistency	`true`
	Findnode lookup consistency	`true`
	Responsibility transfer	`false`
STRUCTURAL	Membership guarantee	`true`
	Reachability	`true`

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 [23]. These results are consistent with previous evaluations using Alloy for the same properties, as discussed in earlier work [22].

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

8

Conclusions and Future Work

Contents

In this chapter, we present our conclusions in Section 8.1, summarizing the main contributions of the work developed, while also outlining potential next steps and highlighting future research avenues in Section 8.2.

8.1 Conclusions

Modern infrastructures — clouds, blockchains, and financial systems — depend on distributed protocols such as 2PC, DHTs, and consensus algorithms [1]. Ensuring their correctness is vital yet difficult: verified algorithms often fail in implementation [4, 9].

Among existing validation methods, such as deductive proofs, model checking, fuzzing, and monitoring, this work focuses on offline monitoring, analyzing complete traces for practical and scalable correctness checking. Existing tools like Alloy and TLA+ [20, 21] lack efficiency at scale.

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

A. **Action Temporal Logic (ACTL): a new action-based temporal logic.** We propose ACTL, a temporal logic tailored for distributed protocols. It results from an iterative refinement of dFOATL, combining the interval-based reasoning of ATL with the expressiveness and quantification power of FOLTL. ACTL models actions as first-class entities — operations with inputs, outputs, and durations — allowing concise specification of temporal, causal, and data-dependent properties. We formally define its syntax and semantics, provide a decision procedure for trace checking, and relate it to dFOATL through a formal encoding.

B. **Formal specification of distributed protocols.** We demonstrate the expressiveness of ACTL by specifying key correctness properties of two canonical distributed protocols: 2PC (atomic commitment) and DHT-based systems (peer-to-peer lookup). These examples illustrate how temporal logics, particularly ACTL, can formally capture safety and consistency constraints across system executions.

C. **ACTLChecker: an efficient monitoring tool.** Implemented in Python, ACTLChecker verifies whether execution traces satisfy ACTL formulas. It includes:

    A. a *Formula Processor*, parsing ACTL specifications into an AST;

    B. a *Log Processor*, transforming instrumented execution logs into structured traces; and

    C. a *Monitor*, which combines both and runs an optimized decision algorithm to determine property satisfaction.

D. **Evaluation on OpenChord traces.** We evaluate ACTLChecker on execution logs from OpenChord, validating multiple DHT properties and disproving one, as expected. Our approach outperforms previous Alloy-based monitors [22] in efficiency and scalability, handling larger traces and offering stronger practical guarantees for protocol monitoring.

8.2 Future Work

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.

Bibliography

[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] J. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Trans. Dependable Secur. Comput.*, vol. 3, no. 3, pp. 202–215, 2006. [Online]. Available: https://doi.org/10.1109/TDSC.2006.35

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, “Verdi: a framework for implementing and formally verifying distributed systems,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, D. Grove and S. M. Blackburn, Eds. ACM, 2015, pp. 357–368. [Online]. Available: https://doi.org/10.1145/2737924.2737958

[11] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. E. Anderson, “Planning for change in a formal verification of the raft consensus protocol,” in *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, J. Avigad and A. Chlipala, Eds. ACM, 2016, pp. 154–165. [Online]. Available: https://doi.org/10.1145/2854065.2854081

[12] 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

[13] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, “Verifying safety properties with the TLA+ proof system,” in *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, J. Giesl and R. Hähnle, Eds., vol. 6173. Springer, 2010, pp. 142–148. [Online]. Available: https://doi.org/10.1007/978-3-642-14203-1_12

[14] 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

[15] 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

[16] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, “Greybox fuzzing of distributed systems,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 1615–1629. [Online]. Available: https://doi.org/10.1145/3576915.3623097

[17] Y. Chen, F. Ma, Y. Zhou, M. Gu, Q. Liao, and Y. Jiang, “Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay,” in *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 2024, pp. 1939–1955. [Online]. Available: https://doi.org/10.1109/SP54263.2024.00109

[18] 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

[19] H. Howard, M. A. Kuppe, E. Ashton, A. Chamayou, and N. Crooks, “Smart casual verification of the confidential consortium framework,” in *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*, T. A. Benson and R. N. Mysore, Eds. USENIX Association, 2025, pp. 259–276. [Online]. Available: https://www.usenix.org/conference/nsdi25/presentation/howard

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] S. Kaffille and K. Loesing, “OpenChord: Implementation of the Chord DHT,” https://open-chord.sourceforge.net/, 2008. [Online]. Available: https://open-chord.sourceforge.net/

[29] 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

[30] ——, “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

[31] 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

[32] T. Braüner and S. Ghilardi, “First-order modal logic,” in *Handbook of Modal Logic*, ser. Studies in logic and practical reasoning, P. Blackburn, J. F. A. K. van Benthem, and F. Wolter, Eds. North-Holland, 2007, vol. 3, pp. 549–620. [Online]. Available: https://doi.org/10.1016/s1570-2464(07)80012-7

[33] A. Artale, A. Mazzullo, and A. Ozaki, “First-order temporal logic on finite traces: Semantic properties, decidable fragments, and applications,” *ACM Trans. Comput. Log.*, vol. 25, no. 2, pp. 13:1–13:43, 2024. [Online]. Available: https://doi.org/10.1145/3651161

[34] L. Geatti, A. Gianola, and N. Gigante, “First-order automata,” in *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, T. Walsh, J. Shah, and Z. Kolter, Eds. AAAI Press, 2025, pp. 14 940–14 948. [Online]. Available: https://doi.org/10.1609/aaai.v39i14.33638

[35] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. [Online]. Available: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[36] G. I. Alkhatib and R. S. Labban, "Transaction management in distributed database systems: The case of oracle's two-phase commit," *J. Inf. Syst. Educ.*, vol. 13, no. 2, pp. 95–104, 2002. [Online]. Available: https://jise.org/Volume13/n2/JISEv13n2p95.html

[37] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," *IEEE Trans. Software Eng.*, vol. 9, no. 3, pp. 219–228, 1983. [Online]. Available: https://doi.org/10.1109/TSE.1983.236608

[38] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[39] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings*, ser. Lecture Notes in Computer Science, R. Guerraoui, Ed., vol. 2218. Springer, 2001, pp. 329–350. [Online]. Available: https://doi.org/10.1007/3-540-45518-3_18

[40] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the XOR metric," in *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, ser. Lecture Notes in Computer Science, P. Druschel, M. F. Kaashoek, and A. I. T. Rowstron, Eds., vol. 2429. Springer, 2002, pp. 53–65. [Online]. Available: https://doi.org/10.1007/3-540-45748-8_5

[41] 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

[42] 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.

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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.

[49] 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

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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.

[55] N. Decker, M. Leucker, and D. Thoma, "Monitoring modulo theories," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 2, pp. 205–225, 2016. [Online]. Available: https://doi.org/10.1007/s10009-015-0380-3

[56] P. Felli, M. Montali, F. Patrizi, and S. Winkler, "Monitoring arithmetic temporal properties on finite traces," in *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, B. Williams, Y. Chen, and J. Neville, Eds. AAAI Press, 2023, pp. 6346–6354. [Online]. Available: https://doi.org/10.1609/aaai.v37i5.25781

[57] M. Faella and G. Parlato, "A unified automata-theoretic approach to $\mathrm{ltl}_{\mathrm{f}}$ modulo theories," in *ECAI 2024 - 27th European Conference on Artificial Intelligence, 19-24 October 2024, Santiago de Compostela, Spain - Including 13th Conference on Prestigious Applications of Intelligent Systems (PAIS 2024)*, ser. Frontiers in Artificial Intelligence and Applications, U. Endriss, F. S. Melo, K. Bach, A. J. B. Diz, J. M. Alonso-Moral, S. Barro, and F. Heintz, Eds., vol. 392. IOS Press, 2024, pp. 1254–1261. [Online]. Available: https://doi.org/10.3233/FAIA240622

[58] 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

[59] 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.

[60] 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

[61] G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[62] 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

[63] 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

[64] 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

[65] D. Liben-Nowell, H. Balakrishnan, and D. R. Karger, "Analysis of the evolution of peer-to-peer systems," in *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, A. Ricciardi, Ed. ACM, 2002, pp. 233–242. [Online]. Available: https://doi.org/10.1145/571825.571863