

Specifying Distributed Hash Tables with Allen Temporal Logic

Nuno Policarpo

Instituto Superior Técnico, University of Lisbon

José Fragoso Santos

INESC-ID & University of Lisbon

Alcino Cunha

INESC TEC & University of Minho

João Leitão

NOVA-LINCS & NOVA School of Science and Technology

Pedro Ákos Costa

NOVA-LINCS & NOVA School of Science and Technology

Abstract—Distributed Hash Tables (DHTs) remain to this day a central component of modern peer-to-peer (P2P) systems, which rely on complex DHT protocols to scale to millions of nodes. The correct operation of DHTs is therefore essential for the proper functioning of these systems. For this reason, formal methods have been employed to model and verify a range of correctness properties of various DHT protocols. However, these verification efforts have focused on protocol-specific properties, such as topological invariants, instead of functional properties. This focus limits our understanding of the precise guarantees offered by each protocol. We propose a protocol-independent axiomatization of DHT properties using Allen Temporal Logic (ATL). To validate our axiomatization, we have implemented it in the Alloy analyser and used our implementation both to establish a number of DHT-derived properties and to generate a set of DHT execution traces that cover an exhaustive list of DHT corner case behaviours.

Index Terms—Formal Verification, Distributed Hash Tables, Protocol Axiomatization, Allen Temporal Logic, Linear Temporal Logic, Alloy.

I. INTRODUCTION

DHTs [1], [2], [3], [4], [5], [6], [7] play a crucial role in today’s peer-to-peer systems [8], [9], [10], [11]. Their behaviour is analogous to that of regular hash tables, mapping keys to unique values. However, in contrast to hash tables, which store all key-value pairs within a single storage unit, DHT key-value pairs are distributed across multiple network nodes, with each node being responsible for a specific subset of keys. Typically, DHT protocols make use of self-organizing topologies, requiring each node to know only a fraction of the other nodes in the network, thereby enabling efficient read and write operations.

The complexity of existing DHT implementations, along with their importance to many relevant P2P systems, has led to the emergence of several lines of work to prove correctness properties of DHT protocols, including Chord [2], [12], [13], Pastry [3], and Kademlia [1]. Among these efforts, P. Zave’s work [14], [15] was particularly successful, having uncovered various errors in Chord’s original specification.

Despite recent successes, the existing formalizations of DHTs [15], [16], [17], [18], [19] suffer from an important limitation: they mainly focus on protocol-dependent topological properties instead of functional properties that DHTs are expected to satisfy. For instance, existing formalizations do not characterize the offered guarantees regarding stored keys and

their associated values, which is the fundamental functionality that motivates the use of DHTs. This limitation makes it impossible to rigorously compare the functional guarantees provided by different DHT designs and protocols.

To improve the formal analysis and verification of DHT protocols, we propose the first *implementation-independent axiomatization of DHT properties*. This axiomatization establishes the set of key properties that DHTs are expected to offer, and enables a formal comparison between the guarantees provided by different DHT protocols.

Axiomatizing the properties of DHTs requires us to reason about the time intervals during which DHT operations occur as well as their interrelations. For example: *if a read operation for a key returns a specific value, then that value must have been previously stored in the DHT*. There are several logics for reasoning about time-spanning operations and their interrelations [20], [21], [22], [23], [24]. In this work, we chose to use Allen Temporal Logic (ATL) [20], [21], because it was specifically designed to compactly express relationships between time intervals.

Although ATL is an established formalism for temporal reasoning [25] used in AI planning, there are currently no systems available for the automatic verification of ATL specifications. Therefore, we developed a new encoding of ATL into First Order Linear Temporal Logic (FO-LTL), which we implemented as an Alloy library [26], [27]. Our ATL Alloy library is also a contribution of this work, since it enables the specification and verification of systems that require reasoning about time-spanning operations, not being, therefore, limited in application scope to DHTs.

The main contributions of this work are the following:

- The first implementation-independent axiomatization of DHT properties that focuses on the functional properties that DHTs are expected to satisfy; and
- An encoding of ATL into FO-LTL implemented in the Alloy formal specification language, that was used to validate our DHT axiomatization.

The structure of this document is as follows: Section II provides an introduction to ATL and discusses several popular DHTs. Section III provides an overview of existing formalization efforts of DHT properties and of logics for reasoning about time-spanning events. Section IV introduces a generic

DHT API and axiomatizes its properties using ATL. Section V describes how ATL was encoded in Alloy and how this encoding was used to develop an Alloy model that captures our DHT axiomatization. Section VI discusses the validation of our DHT axiomatization. Finally, Section VII concludes the document, including some ideas for future work.

II. BACKGROUND

This section gives an introduction to DHTs and ATL.

A. DHT Overview

A DHT is a distributed abstraction that maps keys to values providing a functionality akin to a regular hash table, but where data is distributed across different nodes of the network. Each node is responsible for a subset of keys, and the mapping of keys to nodes is based on hash functions applied to both keys and node identifiers. DHTs use decentralized topologies, where each member knows only a subset of the others. These topologies allow for efficient read and write operations without requiring the participation of all members. In the following, we briefly describe different DHT protocols, focusing on how the network routing information is maintained.

Chord [2], [12] was one of the first DHT protocols to be proposed. It organizes nodes in an ordered ring according to their identifiers, assigning to each node a set of keys for which it is responsible based on the ring topology. To enforce the ring topology, Chord executes various maintenance operations that are capable of restoring the correct topology when nodes enter and/or exit the ring and even when nodes fail.

Pastry [3] was proposed after Chord with the goal of minimizing message travel distance by maintaining in each node information about a greater number of neighbours in the form of a routing table. The idea is to keep more information about nodes that are close and less about more distant nodes. Pastry uses a prefix-based proximity metric, where node identifiers are matched progressively digit by digit.

Tapestry [4] is similar to Pastry, but it relies on hierarchical routing tables. This design provides enhanced flexibility and fault tolerance through increased redundancy, leading to more efficient message routing complexity compared to Pastry.

Kademlia [1] further increases redundancy compared to previous approaches by allowing the same key to be stored in multiple nodes. Analogously to Pastry and Tapestry, Kademlia also uses a logical distance metric, with nodes organizing their view of the identifier space into a binary tree, divided into non-overlapping segments, each covering specific distance ranges from the current node.

CAN [6] organizes nodes and keys in a virtual d -dimensional Cartesian coordinate space, where each node assumes responsibility for a specific zone. Keys are deterministically mapped to points in the Cartesian space using a hash function, and these keys are stored by the node that manages the corresponding zone. To facilitate efficient routing, each node maintains a routing table containing information about close nodes within the Cartesian space, enabling effective communication and routing across the distributed network.

Koorde [5], inspired by Chord, optimizes the trade-off between the number of neighbours maintained by each node and the routing distance in DHTs. It retains Chord's simplicity while leveraging de Bruijn graphs [28] for efficient routing. Koorde maps keys to nodes in an identifier space like Chord, with each node maintaining a tunable number of neighbours that range from two to $O(\log n)$, with n being the total number of nodes. The mechanisms for joining and maintaining the overlay topology are similar to those of Chord but adjusted for de Bruijn graph generation.

Kelips [7] differentiates itself from other DHTs by using a relaxed topology that enables fast and fault-tolerant key-value storage, albeit with increased communication through gossip exchanges [29]. Each node is assigned to an affinity group based on their identifiers and maintains as neighbours one node from each of the other affinity groups and k from its own. Keys are mapped to affinity groups using hash functions.

While the variety of topologies and routing procedures used by these protocols makes it difficult to accurately verify their functional properties and compare their guarantees, we do note that at a functional level (and abstracting minor aspects on APIs exposed by different solutions) all existing solutions provide mechanisms to, locate a node responsible for a given identifier, store a value associated with an identifier, and retrieve the value(s) associated with a given identifier.

B. ATL Overview

ATL [20], [21] is an interval-based temporal logic initially designed to capture the temporal reasoning required for artificial intelligence applications. Its core components are temporal intervals and propositions. Intervals are continuous, occur only once, and can be infinitely divided into sub-intervals. In contrast, propositions may hold at various non-continuous periods, distinguishing them from intervals.

ATL can represent the uncertainty and incomplete information of time-spanning events by specifying the relations between intervals instead of the precise time points in which they occur. These relations are expressed in the form of binary interval predicates and predicates that associate propositions with intervals. Table I summarizes the ATL predicates, which are illustrated in Figure 1. Besides the classical predicates of ATL, we include the additional predicate *Finite*, which distinguishes finite intervals from infinite ones. This predicate is essential for modelling DHTs, as their modelling requires reasoning about non-terminating intervals that cannot be expressed in standard ATL.

Using these interval predicates, we can derive auxiliary predicates to improve the readability of ATL formulas. The $In(i_1, i_2)$ predicate holds when i_1 is contained in i_2 ; put formally:

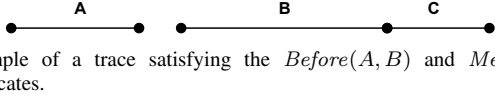
$$In(i_1, i_2) \iff During(i_1, i_2) \vee Starts(i_1, i_2) \vee Finishes(i_1, i_2)$$

The $Requires(i_1, i_2)$ predicate holds when i_1 starts while i_2 is ongoing, i.e. i_1 requires i_2 to be ongoing in order to start;

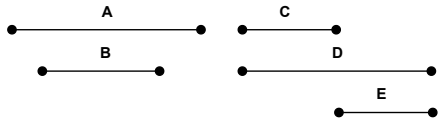
Table I: Summary of ATL predicates.

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

(a) Example of a trace satisfying the $Before(A, B)$ and $Meets(B, C)$ predicates.



(b) Example of a trace satisfying the $During(B, A)$, $Starts(C, D)$, and $Finishes(E, D)$ predicates.



(c) Example of a trace satisfying the $Equal(A, B)$ and $Overlaps(B, C)$ predicates.

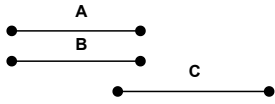


Figure 1: Illustrations of ATL interval predicates

put formally:

$$Requires(i_1, i_2) \iff Equal(i_1, i_2) \vee In(i_1, i_2) \vee Overlap(i_2, i_1) \vee Starts(i_2, i_1)$$

The $Precedes(i_1, i_2)$ predicate holds when i_1 ends before i_2 starts; put formally:

$$Precedes(i_1, i_2) \iff Before(i_1, i_2) \vee Meets(i_1, i_2)$$

The $Intersects(i_1, i_2)$ predicate holds if i_1 and i_2 are ongoing during one common instant; put formally:

$$Intersects(i_1, i_2) \iff Equal(i_1, i_2) \vee In(i_1, i_2) \vee In(i_2, i_1) \vee Overlap(i_1, i_2) \vee Overlap(i_2, i_1)$$

III. RELATED WORK

This section gives a high-level overview of both the verification efforts for popular DHTs and the logics for reasoning about time-spanning events.

A. DHT Verification

P. Zave [14], [15], [16] used Alloy [26] to formally verify Chord's ring-maintenance protocol and assess its correctness claims. Chord's original specification proposed seven invari-

ants [13] that ensured the following correctness property: *if no nodes fail, Chord's maintenance protocol is always able to restore the network's topology to an ideal state, where each node points to its correct successor*. P. Zave produced counterexamples showing violations of the invariants and proposed corrections to the protocol's maintenance operations. As part of this work, P. Zave also proposed a global invariant that guarantees the reachability of an *ideal* state.

T. Lu [17], [30] was the first to formally specify the Pastry protocol, creating a TLA+ [31] model that aimed to verify the correctness of *Lookup* operations. To this end, T. Lu used the proposed model to establish that the closest node to a key is always responsible for that key. However, this property was shown not to hold as concurrent joins could break it. To fix this, T. Lu proposed a modification to the Pastry protocol that limits each node to handle a single join operation at a time. Later, N. Azmy et al. [18], [32] extended T. Lu's work by providing a complete proof that the targeted property holds in a simplified pure-join model.

These verification efforts focused on protocol-specific topological properties, while we aim to specify the functional key-value properties that DHTs are expected to uphold.

B. Logics for Reasoning about Time

Linear Temporal Logic [22] extends classical propositional logic with temporal operators. It was first proposed for the formal verification of computer programs by A. Pnueli [22] in 1977. Propositional Linear Temporal Logic (LTL) is too low-level for specifying complex distributed systems. Extending LTL with first-order quantification can make such specifications much easier. In our case, this is essential since most DHT properties require quantification over the parameters of DHT operations.

Temporal logic model-checkers enable the automatic checking of the satisfiability of LTL formulas. They can be divided into two main groups: bounded model checkers [33] that consider a bounded number of time points, and unbounded model checkers [34], [35] that consider an unbounded number of time points. Although there are many model-checkers for LTL (e.g. [36], [37]), few exist for FO-LTL, the extension of LTL that includes first-order quantification. Examples of model checkers that support FO-LTL are TLC for the TLA+

specification language [31], and the 6th version of the Alloy Analyser which now supports the specification of temporal requirements besides first-order structural properties [26].

Allen Temporal Logic [38] is an *interval-based temporal logic* [38] proposed by J. Allen in 1981 to reason about time-spanning actions in the field of Artificial Intelligence and Planning. Despite its large impact on Computer Science research, for a long time, there was no model-checking technique for ATL. Roşu and Bensalem [39] were the first to show how to verify ATL formulas by encoding them into LTL [22], which required restricting ATL to discrete time. However, the proposed encoding has several limitations: it lacks an implementation; it does not support infinite intervals; and it uses nested temporal operators, which negatively impact performance. As part of this paper, we propose a new encoding of ATL into FO-LTL that addresses these limitations by providing an executable implementation, supporting infinite intervals, and simplifying temporal operators for better solver performance.

IV. AXIOMATIZATION OF DHTS IN ATL

In this section, we introduce a generic DHT API, and specify its desired properties using ATL.

A. DHT API

We present a generic DHT API, which includes most of the operations offered by the state-of-the-art DHTs [1], [2], [3]. We strived to make the API applicable to as many DHTs as possible regardless of their implementation. To achieve this we make minimal assumptions about the topology established between nodes executing the DHT protocol. The assumptions made are that the network configuration consists of uniquely identified nodes and that nodes store key-value pairs; the mapping of keys to nodes is left to the implementation with the only requirement being that the mapping must be deterministic. No further assumptions are made about the configuration information that each node maintains nor the operations used to update and correct this information.

The operations exposed by the API use the following types of parameters: uniquely identified nodes $n \in \mathcal{N}$, keys $k \in \mathcal{K}$, and values $v \in \mathcal{V}$. The generic DHT API exposes the following entry points:

- **store** $(n, k, v) \rightsquigarrow n'$: The client asks node n to add the key-value pair (k, v) to the DHT; the store operation is completed at node n' , which notifies the client when it ends.
- **lookup** $(n, k) \rightsquigarrow (n', v)$: The client asks node n to retrieve the value associated with the key k from the DHT; the lookup operation is completed at node n' , storing the key-value pair (k, v) , which sends the requested value v to the client.
- **findNode** $(n, k) \rightsquigarrow n'$: The client asks node n to find the node n' responsible for storing the key k ; the find operation is completed at node n' , which responds to the client.

- **remove** $(n, k) \rightsquigarrow n'$: The client asks node n to remove the mapping for the key k from the DHT; the removal operation is completed at node n' , which notifies the client when it ends.
- **join** (n) : Node n joins the network, resulting in a configuration in which node n is a member of the network.
- **leave** (n) : 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 node n is no longer a member of the network.
- **fail** (n) : Node n fails without performing maintenance operations, resulting in a new configuration where n is no longer a member of the network.

Note that even though the **lookup** and **findNode** operations do not change the stored mappings, they can still result in new configurations. This is possible because implementations may take advantage of these operations to update routing information as is the case of Pastry and Kademia.

B. DHT Axiomatization in ATL

In this section, we formalize the key properties DHTs should exhibit. Some properties are expected to hold under any conditions while others are expected to only hold under specific network configuration *states* and *regimens*; in such cases we additionally specify those conditions. The use of ATL to formalize DHT properties allows us to reason about the effects of the API operations while abstracting the state of the system, which in this case is the network configuration. More concretely, we specify the properties of DHTs using the ATL predicates applied to API operations, abstracting away implementation details such as the concrete aspects of network configurations.

IV-B1 Component Model We model a generic DHT by defining its core components: nodes, keys, values, and operations. Operations are categorized into two types: functional and membership operations. Functional operations handle keys and values, while membership operations modify the view that nodes have of the network. With the exception of the fail operation, all operations span time. Nonetheless, we model all operations as intervals, with fail operations being represented as instantaneous intervals. We express operation execution in ATL by annotating the operations with the interval i during which they occur. For example, **store** $(n, k, v)_i$ indicates that a **store** operation was executed during interval i where node n initiated the storage of the key-value pair (k, v) . In this expression, the operation may or may not have terminated. To indicate that a functional operation definitely terminated we include the result. The example **store** $(n, k, v) \rightsquigarrow_i n'$ indicates the operation terminated and n' is the node that notifies the client that the operation has concluded.

Besides operations, we also model node membership using intervals, as nodes are members of the DHT for periods of time. For instance, we write **member** $(n)_{i_1}$ to mean that

node n is a member of the network during interval i_1 . As nodes can join and leave the network repeatedly, there may exist k distinct intervals i_1, \dots, i_k during which n is a member of the network, $\mathbf{member}(n)_{i_j}$ with $j = 1, \dots, k$.

For clarity, in the following, we simplify the notation by omitting existentially quantified variables that are not further constrained, using the $-$ symbol. We also abbreviate quantifications as follows:

$$\exists \mathbf{store}(-, k, v) \rightsquigarrow_i n' \equiv \exists k, v, n', i: \mathbf{store}(-, k, v) \rightsquigarrow_i n'$$

Furthermore, quantifiers only apply to variables that have not been previously quantified. In the example below, the second find operation uses the same key k as the first operation because it was already quantified. The node n_2 and interval i_2 are quantified for the first time in the second quantifier and they may have the same values as n_1 and i_1 since nothing restricts it.

$$\begin{aligned} \exists \mathbf{findNode}(-, k) \rightsquigarrow_{i_1} n_1: \\ \exists \mathbf{findNode}(-, k) \rightsquigarrow_{i_2} n_2: \mathit{Before}(i_1, i_2) \end{aligned}$$

As the properties of the **remove** operation largely coincide with those of the **store** operation, we model the **remove** operation as a special **store** operation that associates the special value \perp with the key to be removed. This way we avoid restating the same properties twice.

IV-B2 Operation Preconditions Here, we formally define several basic properties that capture the necessary preconditions for the application of DHT operations. For example, a node cannot join the network if it is already a member. Each operation has preconditions that we detail below to ensure accurate modelling.

a) Non-repeated Operations

The proposed axiomatization establishes that nodes cannot repeat operations that have not yet terminated, i.e., nodes cannot initiate operations if they have previously initiated an identical operation and have not yet received a reply. Below we specify this property for the **store** operation.

$$\boxed{\forall \mathbf{store}(n, k, v)_{i_1}, \mathbf{store}(n, k, v)_{i_2}: i_1 \neq i_2 \implies \mathit{Precedes}(i_1, i_2) \vee \mathit{Precedes}(i_2, i_1)}$$

The property ensures that if node n triggers multiple **store** operations with the same key-value pair, then they do not overlap, i.e., one operation terminates and precedes the other. The non-repeating properties of the remaining operations are identical and, hence, omitted.

b) Membership Operation Preconditions

DHT operations have preconditions related to membership, which we model below.

$$\boxed{\begin{aligned} (1) \forall \mathbf{store}(n, -, -)_{i_1}: \exists \mathbf{member}(n)_{i_2}: \mathit{Requires}(i_1, i_2) \\ (2) \forall \mathbf{join}(n)_{i_1}: \neg \exists \mathbf{member}(n)_{i_2}: \mathit{Requires}(i_1, i_2) \\ (3) \forall \mathbf{leave}(n)_{i_1}: \exists \mathbf{member}(n)_{i_2}: \mathit{Requires}(i_1, i_2) \\ (4) \forall \mathbf{fail}(n)_{i_1}: \exists \mathbf{member}(n)_{i_2}: \mathit{Requires}(i_1, i_2) \end{aligned}}$$

The first property ensures that node n can only start a **store** operation during an interval where it is a member of the DHT. Recall that the predicate $\mathit{Requires}(i_1, i_2)$ holds when i_2 is ongoing when i_1 starts. Conversely, the second property ensures that a node can only join the network if it is not a member. Finally, the third and fourth properties state that a node can only leave the network and fail, respectively, if it is a member.

c) Network State and Runtime Regimens

Certain DHT properties are expected to hold only under specific network conditions. For instance, if two **lookup** operations for the same key are executed, while the network is in an ideal state and no **store** operations are ongoing, then the two lookup operations should return the same value. To specify this type of property, we identify two types of such network conditions: **(1) runtime regimen conditions** that characterize the type of DHT operations that are ongoing at a given instant and **(2) network state conditions** that characterize the network's state at a given instant.

When it comes to *runtime regimen conditions*, we introduce two runtime regimens modeled using the following two interval predicates:

- $\mathit{Stable}(i_1)$ denotes that during interval i_1 no operations that alter the node membership status occur (i.e., no **join**, **leave**, or **fail**); put formally $\mathit{Stable}(i_1)$ is defined by:

$$\begin{aligned} \forall i_2: (\exists \mathbf{join}(-)_{i_2} \vee \exists \mathbf{leave}(-)_{i_2} \vee \exists \mathbf{fail}(-)_{i_2}) \\ \implies \neg \mathit{Intersects}(i_1, i_2) \end{aligned}$$

- $\mathit{ReadOnly}(i_1)$ denotes that during interval i_1 neither membership status nor **store** operations occur (i.e., no **store**, **remove**, **join**, **leave**, or **fail**); put formally $\mathit{ReadOnly}(i_1)$ is defined by:

$$\begin{aligned} \forall i_2: (\exists \mathbf{join}(-)_{i_2} \vee \exists \mathbf{leave}(-)_{i_2} \vee \exists \mathbf{fail}(-)_{i_2} \vee \\ \exists \mathbf{store}(-, -, -)_{i_2}) \implies \neg \mathit{Intersects}(i_1, i_2) \end{aligned}$$

When an interval is not in a *Stable* or *ReadOnly* 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.

When it comes to network state conditions, some DHT properties depend on whether the network is in a state where every node holds sufficient information to forward messages to every other node correctly. We refer to these states as *ideal states* and use the predicate Ideal to model them, i.e., $\mathit{Ideal}(i)$ means that the network is in an ideal state during interval i . The definition of ideal states is implementation-dependent. We consider the network to be in an invalid state if it cannot reach an ideal state through a sequence of implementation-dependent maintenance operations.

The states *Ideal* and *Invalid*, can be seen as the standard two-point lattice $\mathit{Invalid} \leq \mathit{Ideal}$, since all properties that hold in invalid states also hold in ideal states. Similarly, the network regimens, *ReadOnly*, *Stable*, and *Standard*, form a three-point lattice, $\mathit{Standard} \leq \mathit{Stable} \leq \mathit{ReadOnly}$. This means that, assuming no changes in network state, properties that

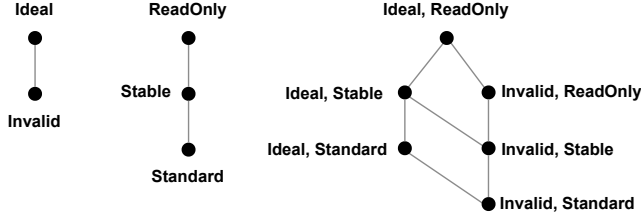


Figure 2: Network state, runtime regimen, and product lattices.

hold in the standard regimen also hold in the stable regimen, and those that hold in the stable regimen also hold in the read-only regimen. Figure 2 illustrates both lattices and the product of the two, resulting in a final lattice that describes the states and regimens relevant for specifying DHT properties.

IV-B3 Functional Properties In the following, we present our axiomatization of the functional properties DHT operations are expected to observe. Table II summarizes these properties and indicates which DHT protocols are supposed to satisfy each property. To make this assessment, we conducted a detailed analysis of the relevant papers, examining the protocol specifications and the algorithmic mechanisms underpinning their design. For instance, the *Responsibility Transfer property* requires protocols to include an explicit leave operation, which is absent in Chord, Pastry, Kademlia, Koorde, and Kelips. In Kelips, however, values associated with keys are shared among the nodes of the same affinity group, resulting in implicit responsibility transfer when a node fails or leaves.

We divide DHT properties into three groups:

- *Value properties* capture guarantees related with the values stored in the DHT;
- *Key properties* reason about the responsibility of nodes with respect to keys;
- *Structural properties* capture aspects related to the network structure and the membership of the DHT.

In the remainder of this section, we explain each property together with its specification. To improve readability, we write $\text{lookup}(-, k, v) \rightsquigarrow_i (n', v) \in R$ to mean that the lookup operation is contained in a network state or regimen R , where $R \in \{\text{Ideal}, \text{Invalid}, \text{ReadOnly}, \text{Stable}\}$. We use the same notation for all other operations.

Property 1 Lookup Consistency. If a **lookup** obtains a key-value pair, then it was previously written by a **store** operation.

$$\boxed{\forall \text{lookup}(-, k) \rightsquigarrow_{i_1} (-, v) : \exists \text{store}(-, k, v)_{i_2} : \text{Precedes}(i_2, i_1) \vee \text{Intersects}(i_1, i_2)}$$

The specification states that if a lookup for key k obtains value v during interval i_1 , then a store of the key-value pair (k, v) occurred during interval i_2 which started before i_1 terminated, meaning that either the store completed before the lookup started (i.e., $\text{Precedes}(i_2, i_1)$) or the two operations occurred concurrently (i.e., $\text{Intersects}(i_1, i_2)$).

Property 2 Value Consistency. In an *Ideal* state during a *ReadOnly* regimen, all **lookup** operations for a given key return the same value.

$$\boxed{\forall \text{lookup}(-, k) \rightsquigarrow_{i_1} (-, v_1), \text{lookup}(-, k) \rightsquigarrow_{i_2} (-, v_2) \in \text{ReadOnly}, \text{Ideal} : v_1 = v_2}$$

The property states that if two lookups for key k occur during an ideal state and read-only regimen, then both operations obtain the same value ($v_1 = v_2$).

Property 3 Value Freshness. In an *Ideal* state all **lookup** operations for a key return the value written by the write operation that most recently terminated, one of its concurrent write operations, or an ongoing write operation.

$$\boxed{\forall \text{lookup}(-, k) \rightsquigarrow_{i_1} (-, v) \in \text{Ideal} : \exists \text{store}(-, k, v)_{i_2} : \text{Intersects}(i_1, i_2) \vee \exists \text{store}(-, k, v)_{i_2} \in \text{MostRecentStores}(i_1) : \text{Precedes}(i_2, i_1)}$$

Where we use $\text{store}(-, k, v)_{i_2} \in \text{MostRecentStores}(i_1)$ to mean that the store during i_2 is one of the most recent stores that precede the interval i_1 , put formally:

$$\forall \text{store}(-, k, -)_{i_3} : \text{Precedes}(i_3, i_1) \implies \neg \text{Precedes}(i_2, i_3)$$

The property states that if a lookup operation returns a value v during an ideal state, then: **(1)** either the lookup is concurrent with a store operation that stored the value v ($\text{Intersects}(i_1, i_2)$), or **(2)** the store operation that stored the value v is one of the most recent stores that terminated before the lookup started ($\text{Precedes}(i_2, i_1)$).

We also define a weaker variant of this property, called **Weak Value Freshness**, that is only required to hold under *ReadOnly* regimens. In an *Ideal* state and a *ReadOnly* regimen all lookup operations return the value written by the most recent write operation.

$$\boxed{\forall \text{lookup}(-, k) \rightsquigarrow_{i_1} (-, v) \in \text{ReadOnly}, \text{Ideal} : \exists \text{store}(-, k, v)_{i_2} \in \text{MostRecentStores}(i_1) : \text{Precedes}(i_2, i_1)}$$

Property 4 Key Consistency. In an *Ideal* state and *Stable* regimen, all members agree on which member is responsible for a key.

$$\boxed{\forall \text{findNode}(-, k) \rightsquigarrow_{i_1} n_1, \text{findNode}(-, k) \rightsquigarrow_{i_2} n_2 \in \text{Ideal}, \text{Stable} : n_1 = n_2}$$

The property states that if two find operations for key k occur during an ideal state and stable regimen, then both operations obtain the same answer ($n_1 = n_2$).

Property 5 FindNode Lookup Consistency. If the lookup of key k returns a value stored by node n , then n must be an obtainable answer to the operation **findNode** of key k .

$$\boxed{\forall \text{findNode}(-, k) \rightsquigarrow_{i_1} n : \exists \text{responsible}(n, k)_{i_2} : \text{Intersects}(i_1, i_2) \vee \forall \text{lookup}(-, k) \rightsquigarrow_{i_1} (n, -) : \exists \text{responsible}(n, k)_{i_2} : \text{Intersects}(i_1, i_2)}$$

Table II: DHT properties, their respective classes, and whether protocols guarantee them.

| Class | Property | Chord | Pastry | Tapestry | Kademlia | CAN | Koorde | Kelips |
|-----------------------|--------------------------------|-------|--------|----------|----------|-----|--------|--------|
| | | [2] | [3] | [4] | [1] | [6] | [5] | [7] |
| Value Properties | 1) Lookup Consistency | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 2) Value Consistency | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ |
| | 3) Value Freshness | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| Key Properties | 4) Key Consistency | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ |
| | 5) FindNode Lookup Consistency | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |
| | 6) Responsibility Expiration | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 7) Responsibility Transfer | ✗ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ |
| Structural Properties | 8) Membership Guarantee | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 9) Reachability | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ |
| | 10) Termination Completeness | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Similarly to the **member** interval, we express that node n is responsible for key k during interval i as **responsible** $(n, k)_i$. This allows us to define properties that explicitly reason about responsibility for keys. The property states that if a find operation during interval i_1 obtains the answer n for key k , then n was responsible for k during an interval i_2 that intersects i_1 . The same applies to a lookup operation that obtains the answer n .

Property 6 Responsibility Expiration. When a node fails and does not rejoin, it eventually stops being responsible for any key.

$$\forall \text{fail}(n)_i : \text{NonJoin}(n, i) \implies \forall \text{responsible}(n, -)_i : \text{Finite}(i)$$

Where we use $\text{NonJoin}(n, i)$ to mean that node n does not join the network after interval i ends, formally:

$$\text{NonJoin}(n, i_1) \iff \forall \text{join}(n)_i : \text{Precedes}(i_2, i_1)$$

The property states that if node n fails during interval i and does not rejoin, then all of its responsibility associations later terminate, therefore eventually it will not be responsible for any key.

Property 7 Responsibility Transfer. When a node leaves the network it immediately ceases to be responsible for any key.

$$\forall \text{leave}(n)_i : \text{NonJoin}(n, i) \implies \forall \text{findNode}(-, -) \rightsquigarrow_{i_2} n : \text{Precedes}(i_2, i) \vee \text{Intersects}(i_1, i_2)$$

In contrast to the previous property, departing nodes are tasked with updating the remaining network and transferring their key responsibilities. The property states that if node n leaves during interval i_1 and does not rejoin, then find operations that return n must have happened before or concurrently with the leave operation.

Property 8 Membership Guarantee. The node returned by a functional operation must have been a member for at least one instant during the execution of the operation.

$$\forall \text{lookup}(-, -) \rightsquigarrow_{i_1} (n, -) : \exists \text{member}(n)_{i_2} : \text{Intersects}(i_1, i_2)$$

The property states that if a lookup operation for key k returns a node n during the interval i_1 . Then n was a member during an interval i_2 such that i_1 and i_2 intersect. This property is a family of axioms that needs to be written for each operation that returns a node, here we presented it for the **lookup** operation.

Property 9 Reachability. If a node n is a member during an *Ideal* state, then all **findNode** operations of the key with the same identifier as node n must return n .

$$\forall \text{findNode}(-, k) \rightsquigarrow_{i_1} n \in \text{Ideal}, \text{member}(n')_{i_2} : k = \text{id}(n') \wedge \text{In}(i_1, i_2) \implies n = n'$$

This property uses the function id which maps nodes to their identifiers. It states that if one performs a **findNode** operation during an ideal state on a key corresponding to the identifier of a member node, then the reply should be the node itself.

Property 10 Termination Completeness. Functional operations that start during an infinite stable regimen terminate.

$$\forall \text{store}(-, -, -)_{i_1, i_2} : \text{In}(i_1, i_2) \wedge \text{Stable}(i_2) \wedge \neg \text{Finite}(i_2) \implies \text{Finite}(i_1)$$

The final property states that in the absence of membership disruptions, every operation terminates. This property is a family of axioms that needs to be written for each functional operation, here we presented it for the **store** operation.

V. ALLOY MODEL

This section describes our encoding of ATL in Alloy and how it was used to develop an Alloy model of our DHT axiomatization.

A. Encoding ATL in Alloy

The main component of ATL are intervals. All intervals must start, so all intervals have a starting boundary, but since they may never end, intervals have one or zero ending boundaries. Intervals are thus defined by their boundaries. In our ATL library we model both intervals and boundaries with signatures, which are used in Alloy to declare sets of entities. To associate intervals with the respective starting and

ending boundaries we use two binary fields, which in Alloy are relations containing pairs of elements, the first belonging to the domain signature (where the field is declared) and the second to the target signature. Field declarations can include a multiplicity keyword that restricts the number of target elements associated with each domain element. The variable signatures `Ongoing` and `Happens` are used to capture the subset of intervals and boundaries that occur at each state along a trace.

```
sig Boundary {}
abstract sig Interval {
  start : one Boundary, end : lone Boundary
}
var lone sig Happens in Boundary {}
var sig Ongoing in Interval {}
```

Two auxiliary functions are used to define the subset of intervals that are currently starting and ending. Starting intervals are those whose start boundary occurs, represented by `start.Happens`, and ending intervals are those whose end boundary occurs, represented by `end.Happens`. Here we used the dot-join composition operator to determine the set of intervals associated with any boundary currently in the subset `Happens`.

```
fun Starting : Interval { start.Happens }
fun Ending   : Interval { end.Happens }
```

Using these functions we can concisely specify the contents of the `Ongoing` intervals with the following fact.

```
fact { always {
  (Starting + Ending) in Ongoing
  Ongoing' = Ongoing - Ending + Starting'
}}
```

Inside the temporal logic operator `always` we first specify that all intervals that are starting or ending (denoted by `Starting + Ending`, where `+` is the set union operator) are also contained in `Ongoing`. Then, we specify the set of ongoing intervals in the next state (denoted by `Ongoing'`, where prime is the operator that determines the next state value of an expression) as the union between current ongoing intervals that do not end (denoted by `Ongoing - Ending`, where `-` is the set difference operator) and those that start in the next state (denoted by `Starting'`).

With the signatures and their rules established, it is now possible to model the interval predicates. Below, we define the `Starts` predicate.

```
pred Starts[i1 : Interval, i2 : Interval] {
  eventually ((i1 + i2) in Starting)
  eventually (i2 in Ongoing and i1 not in Ongoing)}
```

The first expression ensures that i_1 and i_2 start simultaneously, while the second expression ensures that i_1 ends before i_2 .

Our encoding also allows for intervals that span a single time instant (i.e., that start and end at the same state), which we refer to as singletons. The predicate `Singleton` holds when an interval is a singleton.

```
pred Singleton [i : Interval] { i.start = i.end }
```

B. DHT Component Model in Alloy

To translate our DHT axiomatization into Alloy we first declare signatures to model keys, values, and nodes. For simplicity, nodes and keys are represented by their corresponding identifiers. Since they are of the same type, we opted to define node identifiers as an extension of key identifiers, meaning that some key identifiers are also node identifiers.

```
sig Key, Value {}
sig Node extends Key {}
```

Each DHT operation is modeled as a signature that extends the `Interval` signature, effectively identifying each operation by the interval during which it occurs. To simplify the model, we group functional operations into an abstract signature `FunctionalOperation` and membership operations into an abstract signature `MembershipOperation`. These signatures hold all the attributes that the respective operations have in common. The `FunctionalOperation` abstract signature is declared as follows:

```
abstract sig FunctionalOperation extends Interval {
  node, key, replier : one Node
}{ not Singleton[this] }
```

This signature has three field declarations: `node`, representing the node on which the operation is invoked; `key`, representing the target key of the operation; and `replier` representing the replier node. Furthermore, we require that the interval associated with each functional operation is not a singleton (i.e., it must span more than one time instant) via the signature constraint `not Singleton[this]`.

The `store`, `lookup`, and `findNode` functional operations are then declared as extensions of `FunctionalOperation`:

```
sig Store, Lookup extends FunctionalOperation {
  value : one Value
}
sig FindNode extends FunctionalOperation {}
```

Note that the signatures `Store` and `Lookup` have an additional field `value` representing the value being stored or looked up.

The `MembershipOperation` abstract signature is declared as follows:

```
abstract sig MembershipOperation extends Interval {
  node : one Node
}
```

It has a single field `node`, representing the node on which the operation is invoked. In contrast to functional operations, not all membership operations are required to be non-singleton intervals. In particular, the `fail` operation spans a single time instant. Hence, the membership operations that are required to span multiple instants must explicitly include that constraint.

```
sig Join, Leave extends MembershipOperation {} {
  not Singleton[this]
}
sig Fail extends MembershipOperation {} {
  Singleton[this]
}
```

Lastly, we also model membership status, regimens, and states as intervals:

```

sig Member extends Interval { node : one Node }
sig Ideal, ReadOnly, Stable extends Interval{}

```

A `Member` interval represents an interval during which its corresponding `node` is a member of the DHT.

C. Modelling DHT Properties in Alloy

Basic properties can be translated directly to facts using the ATL predicates provided by the library described above. As an example, consider the Alloy specification of the non-repeated operations property. We specified this property for the store operation as:

$$\forall \text{store}(n, k, v)_{i_1}, \text{store}(n, k, v)_{i_2} : \\ i_1 \neq i_2 \implies \text{Precedes}(i_1, i_2) \vee \text{Precedes}(i_2, i_1)$$

In Alloy it is specified as follows:

```

fact { all disj i1, i2 : Store {
  { i1.node = i2.node
    i1.key = i2.key
    i1.value = i2.value
  } implies
  (Precedes[i1, i2] or Precedes[i2, i1]) }
}

```

The Alloy fact is an almost direct encoding of the property and ensures that for every pair of different store operations with the same arguments, one operation precedes the other. Note that the requirement that $i_1 \neq i_2$ is specified by the `disj` modifier on the universal quantifier.

Since operations that share the same properties are grouped into abstract signatures we can specify some properties just once for the abstract signature, instead of repeating them for each operation. For example, the membership requirement property was previously formalized for `store` as follows:

$$\forall \text{store}(n, -, -)_{i_1} : \exists \text{member}(n)_{i_2} : \text{Requires}(i_1, i_2)$$

In Alloy we can generalize this property for all functional operations by quantifying over the abstract signature `FunctionalOperation`:

```

fact { all i1 : FunctionalOperation {
  some i2 : Member {
    i1.node = i2.node and Requires[i1, i2]
  } }
}

```

VI. EVALUATION

This section evaluates our ATL encoding and DHT axiomatization, aiming to address the following two evaluation questions: (1) Does the DHT axiomatization cover the main classes of DHT interactions present in current implementations? (2) Does the DHT axiomatization prevent undesirable behaviours that violate fundamental DHT properties? To answer these questions we relied on a combination of two techniques: scenario exploration and verification of derived properties. The analysis was conducted using the MiniSAT solver with Alloy's bounded model-checking engine, with scopes of 10 intervals, 15 boundaries, 5 keys, and 5 values. Results were obtained on a machine with 16 GB RAM and an Intel Core i7-8565U CPU. Times are reported in seconds.



Figure 3: Alloy generated trace of scenario S4.

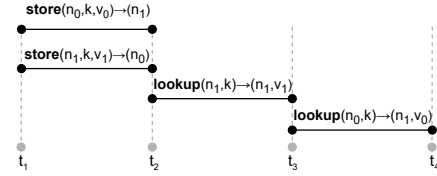


Figure 4: Abstract representation of Alloy generated trace of scenario S4.

A. Scenario Exploration

We used our Alloy DHT model to describe a range of valid DHT execution scenarios and checked whether Alloy could generate execution traces corresponding to each of them. The specified scenarios were the following:

- S1:** All nodes initiate at least one operation;
- S2:** All nodes initiate at least one operation that terminates;
- S3:** All nodes either fail or initiate at least one operation;
- S4:** Concurrent lookup operations return different values;
- S5:** Every type of operation is executed;
- S6:** Every type of functional operation is executed and terminates;
- S7:** A lookup operation reads a value written by a concurrent store operation;
- S8:** A node that leaves the network is later the returned node of a find node operation;
- S9:** A node fails and is later returned as the answer to a find node operation;
- S10:** A find node operation is performed using a key that matches the identifier of a node that previously left.

As expected, Alloy successfully generated valid traces for all of the above scenarios. Table III reports the time taken by Alloy to generate each scenario and the required minimum trace length.

Table III: Alloy Analyzer performance in scenario exploration.

| Scenario | Time (s) | Steps |
|------------|----------|-------|
| S1 | 49 | 2 |
| S2 | 48 | 3 |
| S3 | 98 | 3 |
| S4 | 102 | 3 |
| S5 | 99 | 3 |
| S6 | 110 | 4 |
| S7 | 98 | 3 |
| S8 | 165 | 4 |
| S9 | 168 | 4 |
| S10 | 254 | 5 |

Figure 3 depicts a trace with five steps generated for the fourth scenario, where two lookup operations obtain different values for the same key. Figure 4 illustrates an abstract representation of the same trace. To improve readability, we represent the starting intervals in green, the ending intervals in red, and the remaining intervals in white. The whole trace occurs in an ideal state, where each member can send messages to every other member, and during a stable regimen, where nodes do not exit nor join the network. The generated trace is as follows: **(1)** At instant t_1 , node n_0 initiates a store operation of the key-value pair (k, v_0) which terminates at instant t_2 with a reply from n_1 . **(2)** Concurrently, at instant t_1 , node n_1 initiates a store operation of the key-value pair (k, v_1) which terminates at instant t_2 with a reply from n_0 . **(3)** At instant t_2 , node n_1 initiates a lookup operation for k which terminates at t_3 obtaining value v_1 ; **(4)** At instant t_3 , node n_0 initiates a lookup operation for key k which terminates at t_4 obtaining value v_0 . This trace captures the fact that if two store operations for the same key overlap, then either of the two corresponding values may be returned by subsequent lookup operations.

These results show that our DHT axiomatization can be used to generate non-trivial workloads to test real DHT implementations.

B. Derived Assertions

As a second way of validating our axiomatization, we devised a number of assertions that must be satisfied by any DHT protocol satisfying our axiomatization and used Alloy to verify their bounded correctness. More specifically, we instructed Alloy to generate counter-examples for the following assertions:

A1: No functional operations are initiated if there are no members;

A2: During a global ideal state, lookups do not return stale values;

A3: Value Freshness implies Weak Value Freshness;

A4: If the network is always in the ideal state, then Value Freshness implies Lookup Consistency;

As expected, Alloy could not find counter-examples for the assertions above, thereby establishing their bounded correctness.

Table IV: Alloy Analyser performance checking derived assertions.

| Assertion | Time (s) | | |
|-----------|----------|----------|----------|
| | 10 steps | 15 steps | 20 steps |
| A1 | 775 | 1375 | 2410 |
| A2 | 649 | 1387 | 2352 |
| A3 | 100 | 321 | 712 |
| A4 | 122 | 183 | 328 |

Table IV reports the time required to check these assertions for 10, 15, and 20 steps. As expected, the time increases significantly as the number of steps grows.

VII. CONCLUSION

DHTs are essential for large-scale decentralized peer-to-peer systems, enabling node discovery and efficient and effective data storage. To reason about their correctness, we proposed an implementation-independent axiomatization focusing on the functional guarantees DHTs are expected to provide. This axiomatization allows systematic reasoning and facilitates comparisons between the guarantees offered by different DHT designs and protocols. To model this axiomatization we developed a new encoding from Allen Temporal Logic into FO-LTL, implemented as an Alloy library. This library enables the specification and verification of systems that involve time-spanning events, extending its utility beyond just DHTs. The proposed ATL to FO-LTL encoding and the DHT axiomatization are available as two separate open-source projects.¹

As future work, we plan to analyse concrete DHT implementations, evaluating which axiomatization properties they uphold. To achieve this we will instrument the targeted DHT implementations to log node operations and use the generated logs to check which axiomatisation properties are satisfied.

Acknowledgements

We thank the anonymous reviewers for their comments and feedback. This work was partially supported by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, through projects LA/P/0063/2020, UID/04516/NOVA, UIDB/50021/2020, and 2024.07393.IACDC (WebCAP), by IAPMEI via project C6632206063-00466847 (PT Smart Retail), and by EU Horizon Europe under Grant Agreement no. 101093006 (TaRDIS).

REFERENCES

- [1] P. Maymounkov and D. Mazières, “Kademlia: A peer-to-peer information system based on the xor metric,” in *Peer-to-Peer Systems*, P. Druschel, F. Kaashoek, and A. Rowstron, Eds., vol. 2429. Berlin, Heidelberg: Springer Berlin Heidelberg, April 2002, pp. 53–65.
- [2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *Proceedings of SIGCOMM*, vol. 31, August 2001.
- [3] A. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems,” vol. 2218, 01 2001, pp. 329–350.

¹<https://github.com/NNate1/ATL>

- [4] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.
- [5] M. F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal distributed hash table," in *Peer-to-Peer Systems II*, M. F. Kaashoek and I. Stoica, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 98–107.
- [6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 161–172. [Online]. Available: <https://doi.org/10.1145/383059.383072>
- [7] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse, "Kelips: Building an efficient and stable p2p dht through increased memory and background overhead," in *Peer-to-Peer Systems II*, M. F. Kaashoek and I. Stoica, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 160–169.
- [8] S. Sarmady, "A survey on peer-to-peer and dht," 06 2010.
- [9] D. Tracey and C. Sreenan, "Using a dht in a peer to peer architecture for the internet of things," in *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, 2019, pp. 560–565.
- [10] J. Benet, "Ipfis - content addressed, versioned, p2p file system," 2014. [Online]. Available: <https://arxiv.org/abs/1407.3561>
- [11] J. Monteiro, P. Á. Costa, J. Leitão, A. De la Rocha, and Y. Psaras, "Enriching kademia by partitioning," in *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2022, pp. 33–38.
- [12] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE Transactions on Networking*, vol. 11, no. 2, pp. 2003.
- [13] D. Liben-Nowell, H. Balakrishnan, and D. Karger, "Analysis of the evolution of peer-to-peer systems," *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 08 2004.
- [14] P. Zave, "Using lightweight modeling to understand chord," *Computer Communication Review - CCR*, vol. 42, pp. 49–57, 03 2012.
- [15] —, "Reasoning about identifier spaces: How to make chord correct," *IEEE Transactions on Software Engineering*, vol. PP, 10 2016.
- [16] —, "A practical comparison of alloy and spin," *Formal Aspects of Computing*, vol. 27, 03 2014.
- [17] T. Lu, "Formal verification of the pastry protocol using tla+," in *Dependable Software Engineering: Theories, Tools, and Applications*, X. Li, Z. Liu, and W. Yi, Eds. Cham: Springer International Publishing, 2015, pp. 284–299.
- [18] N. Azmy, S. Merz, and C. Weidenbach, "A machine-checked correctness proof for pastry," *Science of Computer Programming*, vol. 158, 08 2017.
- [19] I. Pita and A. Riesco, "Specifying and analyzing the kademia protocol in maude," in *Theoretical Aspects of Computing - ICTAC 2015*, M. Leucker, C. Rueda, and F. D. Valencia, Eds. Cham: Springer International Publishing, 2015, pp. 524–541.
- [20] J. F. Allen, "Towards a general theory of action and time," *Artificial Intelligence*, vol. 23, no. 2, pp. 123–154, 1984.
- [21] —, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, p. 832–843, nov 1983. [Online]. Available: <https://doi.org/10.1145/182.358434>
- [22] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.
- [23] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994.
- [24] R. Kowalski and M. Sergot, "A logic-based calculus of events," *New Generation Computing*, vol. 4, pp. 67–95, 01 1985.
- [25] A. Krokhin, P. Jeavons, and P. Jonsson, "Reasoning about temporal relations: The tractable subalgebras of allen's interval algebra," 08 2001.
- [26] D. Jackson, *Software Abstractions: logic, language and analysis*, 01 2006.
- [27] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 373–383. [Online]. Available: <https://doi.org/10.1145/2950290.2950318>
- [28] N. de Bruijn, "A combinatorial problem," *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen*, vol. 49, pp. 758–764, 1946.
- [29] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '87. New York, NY, USA: Association for Computing Machinery, 1987, p. 1–12. [Online]. Available: <https://doi.org/10.1145/41840.41841>
- [30] T. Lu, S. Merz, and C. Weidenbach, "Towards verification of the pastry protocol using tla+," in *Formal Techniques for Distributed Systems*, R. Bruni and J. Dingel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 244–258.
- [31] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto, "Tla+ proofs," in *FM 2012: Formal Methods*, D. Gianakopoulou and D. Méry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 147–154.
- [32] N. Azmy, S. Merz, and C. Weidenbach, "A rigorous correctness proof for pastry," 05 2016, pp. 86–101.
- [33] D. Kroening and M. Tautschnig, "Cbmc - c bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Abraham and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391.
- [34] L. Geatti, N. Gigante, and A. Montanari, "A sat-based encoding of the one-pass and tree-shaped tableau system for LTL," in *Proceedings of the 28th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, ser. Lecture Notes in Computer Science, S. Cerrito and A. Popescu, Eds., vol. 11714. Springer, 2019, pp. 3–20.
- [35] L. Geatti, A. Gianola, N. Gigante, and S. Winkler, "Decidable fragments of Itl_f modulo theories," in *ECAI 2023 - 26th European Conference on Artificial Intelligence*, September 30 - October 4, 2023, Kraków, Poland - Including 12th Conference on Prestigious Applications of Intelligent Systems (PAIS 2023), ser. Frontiers in Artificial Intelligence and Applications, K. Gal, A. Nowé, G. J. Nalepa, R. Fairstein, and R. Radulescu, Eds., vol. 372. IOS Press, 2023, pp. 811–818. [Online]. Available: <https://doi.org/10.3233/FAIA230348>
- [36] G. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, 01 2004.
- [37] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *STTT*, vol. 2, pp. 410–425, 03 2000.
- [38] J. F. Allen, "An interval-based representation of temporal knowledge," in *IJCAI*, vol. 81. Citeseer, 1981, pp. 221–226.
- [39] G. Roşu and S. Bensalem, "Allen linear (interval) temporal logic – translation to ltl and monitor synthesis," in *Computer Aided Verification*, T. Ball and R. B. Jones, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 263–277.