Mechanically Verifying the Fundamental Liveness Property of the Chord Protocol

Jean-Paul Bodeveix[1], Julien Brunel[2], David Chemouil[2], and Mamoun Filali[1]

[1] IRIT CNRS UPS, Université de Toulouse, France, first.last@irit.fr
[2] ONERA DTIS, Université de Toulouse, France, first.last@onera.fr

Abstract. Chord is a protocol providing a scalable distributed hash table over an underlying peer-to-peer network. It is very popular due to its simplicity, performance and claimed correctness. However, the original version of the Chord *maintenance* protocol, presented with an informal proof of correctness, was since then shown to be in fact incorrect. It is actually tricky to come up with a provably-correct version as the protocol combines data structures, asynchronous communication, concurrency, and fault tolerance. Additionally, the correctness property amounts to a form of *stabilization*, a particular kind of liveness property. Previous work only addressed automated proofs of safety; and pen-and-paper, or automated but much bounded, proofs of stabilization. In this article, we report on the first mechanized proof of the liveness property for Chord. Furthermore, our proof addresses the full parameterized version of the protocol, weakens previously-devised invariants and operating assumptions, and is essentially automated (requiring limited effort when manual assistance is needed).

Keywords: Chord · Distributed protocol · Parameterized verification · Liveness · Stabilization proof.

1 Introduction

Chord [10,17,18] is a popular distributed lookup protocol addressing an essential issue of peer-to-peer applications: efficiently localizing some sought data in a dynamically-evolving network. To achieve this, the Chord protocol is designed so as to maintain a *ring* topology, as much as possible, and to fix possible disruptions due to nodes joining or leaving the network, or failing. When it was first introduced, Chord was claimed to be simple, efficient and correct. However, Zave [20] identified some flaws in the *maintenance protocol* (the only aspect we consider in this paper) and proposed some corrections. Since then, Chord has been used as a test-bed for various formal studies [3, 5, 15, 19], using various methods and languages, including an outstanding endeavor by Zave herself [20–23]. However, most work has focused on proofs of safety while the fundamental correctness property of Chord is a *stabilization* property, a particular kind of liveness property, saying that *if, from a certain instant, there is no subsequent join, departure or failure, then the network is ensured to recover a ring topology eventually and to keep it.*

In her work [23], Zave identified key invariants that are instrumental to make the proof of liveness doable. She was able to check them using Alloy [7] but had to resort to good old pen and paper to provide a proof of liveness (unachievable in Alloy). In [5], some of the authors of the present paper used Electrum [11], a temporal extension of Alloy, to address the liveness proof in an automated way but only for networks of small size.

In this paper, we present a proof of correctness (liveness property) of the Chord maintenance protocol with the following contributions:

- our proof is parametric in the number of nodes in the network and in the number of redundant data used for robustness (so-called "successor lists", cf. section 2.1.2);
- we address the problem in a mechanized setting and rely on various abstractions so that most proof obligations are automatically discharged while most manual proofs need only limited manual intervention;
- we develop a proof method to address the specific shape of the liveness property at stake;
- we show that several invariants and operating assumptions made in the literature can be logically weakened.

Our work is performed using the Event-B language [1] and the accompanying tool Rodin [2]. We first use superposition refinement (also called *horizontal refinement*) to build the protocol incrementally. Then technical refinements are introduced to make Rodin produce the wanted proof obligations for stabilization. Thus, we do not really follow a refinement-based method to derive a correct protocol. Rather, we rely on Event-B and Rodin to take advantage of the ability to write specifications in an expressive language that the built-in pivot solver can translate and forward to SMT solvers, with great success in most cases for this work. For this reason, this article is written with the aim of presenting the essential aspects of our approach. Full Event-B models can be found at [4].

In section 2, we present our model of the Chord maintenance protocol and describe our proof methodology. Then, in section 3, we address properties of a Chord network, showing in particular how known operating assumptions and invariants can be weakened. In section 4, we show that the maintenance protocol ensures the liveness property presented above. Finally we present related work in section 5 and discuss future work in section 6.

2 The Chord Protocol

This section presents the Chord network topology forming a ring, Chord data and the protocol itself as a set of guarded symbolic transitions.

2.1 Network Structure

2.1.1 Identifier Space In a Chord network, every node has an identifier (a hash of its IP address). Pairs of keys and associated data are stored in nodes. In

this article, we conflate the notions of a node and its identifier, and thus use a set NODE of node (identifiers).

The *node identifier space* is structured as a ring-shaped directed graph. Intuitively, identifiers are ordered following the usual strict ordering on natural numbers (written $<$ in the following), wrapping around at the largest identifier in order to close the ring. Due to this shape, situating an identifier is advantageously modeled by checking whether it sits *between* two other identifiers: given $n_1, n_2 \in$ NODE, we define the *set of identifiers between* n_1 *and* n_2, written $n_1 \rightsquigarrow n_2$, by[3]:

$$n_1 \rightsquigarrow n_2 \triangleq \left\{ n \in \text{NODE} \;\middle|\; \begin{array}{ll} n_1 < n < n_2 & \text{if } n_1 < n_2 \\ n_1 < n \text{ or } n < n_2 & \text{otherwise} \end{array} \right\}$$

Given a node $n \in$ NODE, we note $\texttt{next}(n)$ the next node according to $\rightsquigarrow$, *i.e.*, s.t. $n \rightsquigarrow \texttt{next}(n) = \emptyset$.

2.1.2 Chord Network A Chord *network* is thus built over the identifier space. In order for Chord to provide an efficient lookup procedure, *ideally*, the network should also form a ring-shaped digraph at every instant, where every member is in charge of storing some payload (depending on the node identifier) and points to its *nearest successor* among the ring members (see fig. 1 (left)).

However, as nodes dynamically join and leave the network, this ideal ring shape cannot always be maintained and *appendages* to the ring will appear (see fig. 1 (middle)). The set of nodes belonging to the Chord network, that is belonging to the Chord ring or to its appendages, is called MEMBERS. Its elements are also called *live* nodes. Non-members are called *dead* nodes. Formally, a node may be dead either because it was live and later failed or left the network, or because it never joined the network.

Thus, the protocol is in fact meant to keep the network in a repairable state and, in the long run, to fix disruptions.

To enhance robustness to failures, every live node holds a fixed-length *successor list* [17, Sect. 5.2] of K pointers to other nodes[4], where K is a parameter of the protocol. This way, if a node leaves the network, its predecessor will still have successors in the network: an assumption is made, stating that every node always has at least one live node in its successor list (see Sect. 3.2.1). Additionally, every live node also holds a (possibly-null) pointer to its *predecessor* node: this is useful in the execution of maintenance operations.

Thus we end up with the following state variables (`prdc` is declared as a partial function as it may be undefined for some members).

$$\text{MEMBERS} \subseteq \text{NODE} \qquad \texttt{succ} : \text{MEMBERS} \to \texttt{list}_K(\text{NODE})$$
$$\texttt{prdc} : \text{MEMBERS} \nrightarrow \text{NODE}$$

[3] In our Event-B model, we actually use a pure first-order axiomatization, presented *e.g.* in [15], which allows SMT solvers to deal with many proofs automatically.

[4] Not to be confused with Chord's *finger tables* whose purpose is to support efficient query routing [17, Sect. 4.3].

We also use `bestSucc` : MEMBERS → MEMBERS to indicate the first live node among the K successors of a member: $\texttt{bestSucc}(n) \triangleq \texttt{succ}(n)[i]$ where i is the least index s.t. $\texttt{succ}(n)[i]$ is alive.

2.2 Chord Operations

2.2.1 Formal Model We present (end of section 2.2) the Chord operations using a pseudo-code reminiscent of classic formal specification languages. In practice, we relied on the Event-B notation, essentially because we wanted to use the accompanying tool Rodin as a pivot solver for a specification which is parametric in the number of nodes and the length of successor lists. The meaning of our notation (which is *indentation-dependent* for brevity) is as follows:

- events have parameters, a guard (introduced by the keyword **`guard`**), which is a conjunction of formulas, and an action body (introduced with **`do`**);
- the execution of the action part of an event is *atomic* and consists in the *simultaneous* execution of all its statements (no sequentiality in actions);
- *interleaving*: at every instant, a single event is fired (proof obligations check that at least one event can be fired at every instant);
- as in Event-B, we do not make any by-default fairness assumption on the execution model, but our proof will suppose strong fairness on events (see Section 2.3);
- instead of using classic function application on nodes (*e.g.* `prdc(n)`), we use the dot notation (`n.prdc`) to emphasize that the variables we consider can be seen as node fields (`n.f` = $\perp$ states that f is undefined for n);
- contrary to Event-B, we also have a notion of conditional, where every **`then`** or **`else`** branch is *tagged* with a label starting with the `@` symbol: this provides a concise way to describe several Event-B events at once[5].

Modeling-wise, following [23], an event corresponds to an operation executed by a single node. It may communicate with only one, other node; and there is a time-out such that it allows nodes to detect live or dead nodes. These rules aim at faithfully abstracting the distributed system that Chord is.

2.2.2 Model-Specific State Variables Apart from the previously-mentioned state variables, our model of the protocol also features two further state variables `Stabilizing` and `Rectifying`:

$$\texttt{Stabilizing} : \text{MEMBERS} \nrightarrow \text{NODE} \qquad \texttt{Rectifying} \subseteq \text{NODE} \times \text{NODE}$$

The former is used to model the fact that, while an operation, called *stabilize* in Chord, is running on a live node, some significant state changes may happen

[5] An event `E` with guard `g` and body **`if`** `c` **`then`** `@1 t` **`else`** `@2` **`(if`** `c2` **`then`** `@a t2` **`else`** `@b e)` will give rise to an Event-B event `E1` with guard `g` **`and`** `c` and body `t`, and to Event-B events `E2a` (resp. `E2b`) with guard `g` **`and not`** `c` **`and`** `c2` and body `t2` (resp. with guard `g` **`and not`** `c` **`and not`** `c2` and body `e`).

elsewhere. In our model, as in [23], this operation is split into two in order to allow this "preemption", and if `Stabilizing` is defined for a given member `m`, then `m.Stabilizing` yields its memory context (the stored identifier of another node).

`Rectifying` is here to account for asynchronous communication. In some contexts, a node may send a message to another node to tell the latter to perform a so-called *rectification*. Intuitively, this binary relation associates a node with the set of messages it has sent and that have not been handled yet. Notice the type of `Rectifying`: we do not consider the order in which messages are sent or received, nor the duplication of messages from a given node to another one. Additionally, not restricting the domain of `Rectifying` to MEMBERS allows us to model a message to a node which has failed since the message was sent.

2.2.3 Events The Chord operations[6], shown later, follow the presentation by Chord authors in [10,17], P. Zave [23] and some authors [5] of the present article, with a few variations.

The first two events are `join` and `fail` and are under the control of the environment. The `fail` event models a failure or a voluntary departure. Notice that an operating assumption on `fail` is necessary and presented in section 3.2.1.

In the case of the `join` event, a `new` node can join the Chord network by taking a well-positioned live node `m` as its predecessor and taking `m`'s successor list as its list too. Lines 9 to 10, which concern the `Rectifying` field, are here to model a special situation: as explained above, `Rectifying` represents asynchronous communication. When a node `m` sends a `Rectifying` message to a node `n`, `n` may: (1) receive it (and handle it), (2) fail and therefore miss it, *or* (3) fail and join again fast enough to still receive it. To account for this distributed aspect, instead of modeling a channel explicitly, we keep our simple modeling with the following specificity: when a (previously failed) node joins, *some* `Rectifying` messages addressed to it are chosen non-deterministically and lost (line 9).

Maintenance operations aim at compensating disruptions due to nodes joining and failing. The first such operation is *stabilization*. Its purpose is to fix the first successor of a node. As explained in section 2.2.2, to account for possible state changes during its execution, the operation is split into two as in [23]. The first part, `stabilizeFromFst`, can only happen on a live node if it is not already doing a stabilization (line 15). The node first checks whether its first successor is live. If not, the node updates its successor list by shifting it one step to the left, and padding it at the end with the lowest identifier following its last known successor (line 19). There may be no node corresponding to this identifier but, as it is the lowest possible, it prevents skipping possible live nodes and it can eventually be fixed. Otherwise, the successor list is just updated with fresh data coming from its first successor (line 22). Finally, the node checks whether its first successor's predecessor is better placed than itself. In this case, it decides to update its first successor: as explained above, stabilization is not over yet but, to account for possible changes in parallel, we just memorize that it should

[6] We write ++ (resp. `::`) for list concatenation (resp. cons), and `x -= s` (resp. `x += s`) for `x := x \ s` (resp. `x := x ∪ s`). Abusing notation, a singleton set `{s}` is written `s`.

continue the operation later with this better successor (line 24). Otherwise, the first successor is sent a message saying that it should update its predecessor.

The second stabilization part, `stabilizeFromFstPrdc`, precisely continues the operation. It can only be fired if the `Stabilizing` field is non-null, in which case it holds a well-located, candidate new value for the first successor. Yet, as changes may have happened, this node is tested for being a member. If it is dead, there is nothing to do: the operation is over, the current successor is just sent a message to tell it to update its predecessor. Otherwise, the candidate node is taken as a new successor and similarly asked to update its predecessor.

Finally, *rectification* aims at fixing predecessor pointers. The **`rectify`** operation consumes a message sent during the stabilization of a candidate predecessor. If the current predecessor is dead or if the candidate is nearer than the current one, then an update of the predecessor pointer is done, otherwise nothing happens. Finally, the **`rectifyNull`** operation can be spontaneously fired. It sets the predecessor pointer to null if the pointed node is dead.

```
1  event join [new, m]
2  guard
3    new ∉ MEMBERS
4    m ∈ MEMBERS
5    new ∈ m ⇝ m.succ[1]
6  do // atomic
7    new.succ := m.succ
8    new.prdc := m
9    choose loss ⊆ NODE then
10     new.Rectifying −= loss
11
12 event stabilizeFromFst [m]
13 guard
14   m ∈ MEMBERS
15   m.Stabilizing = ⊥
16 do // atomic
17   if m.succ[1] ∉ MEMBERS then @1
18     m.succ :=
19       tail(succ) ++ next(m.succ[K])
20   else @2
21     m.succ :=
22       m.succ[1] :: butLast(m.succ[1].succ))
23     if m.succ[1].prdc != ⊥
24       and m.succ[1].prdc ∈ m ⇝ m.succ[1]
25     then @sta
26       m.Stabilizing := m.succ[1].prdc
27     else @rct
28       m.succ[1].Rectifying += m
29
30 event rectifyNull [m]
31 guard
32   m ∈ MEMBERS
33   m.prdc != ⊥
34   m.prdc ∉ MEMBERS
35 do // atomic
36   m.prdc := ⊥
37 // fail is also subject to an operating
38 // assumption (cf. section 3.2.1)
39 event fail [f]
40 guard
41   f ∈ MEMBERS
42 do
43   MEMBERS −= f
44
45
46
47
48 event stabilizeFromFstPrdc [m, newFst]
49 guard
50   m ∈ MEMBERS
51   m.Stabilizing = newFst
52   newFst ∈ m ⇝ m.succ[1]
53 do // atomic
54   if newFst ∉ MEMBERS then @1
55     m.Stabilizing := ⊥
56     m.succ[1].Rectifying += m
57   else @2
58     m.succ = newFst :: butLast(newFst.succ)
59     m.Stabilizing := ⊥
60     newFst.Rectifying += m
61
62 event rectify [m, newPrdc]
63 guard
64   m ∈ MEMBERS
65   newPrdc ∈ m.Rectifying
66 do // atomic
67   if m.prdc ∉ MEMBERS
68       or newPrdc ∈ m.prdc ⇝ m then @1
69     m.Rectifying −= newPrdc
70     m.prdc := newPrdc
71   else @2
72     m.Rectifying −= newPrdc
```

2.3 Proof Engineering

The proofs of this article have been mechanized thanks to the Rodin framework. The framework is here used as a proof obligation generator and as an environment to discharge generated proofs (through user interaction). The framework contains built-in solvers and is also connected to external SMT solvers. The basic machinery available within Rodin allows for the automatic generation of proof obligations for invariants, event convergence, refinements and theorems. An invariant property is true initially and preserved by each event. Event convergence is established through the introduction of a variant which is an expression yielding a natural number or a finite set. Each *convergent* event must decrease the variant strictly. Event-B also provides *anticipated* events which must not increase the variant. We use these features to generate proof obligations for stabilization.

Since the main property of Chord is stabilization under the hypothesis that the events `join` and `fail` do not occur anymore and strong fairness [8] over the other events E in a context H, we propose here proof obligations for establishing the stabilization of a given property Q[7]:

1. $\bigwedge_{e \in E \setminus C} H \wedge V = v \Rightarrow [e](V \preceq v)$ (generated by Rodin for *anticipated* events): anticipated events do not increase the variant;
2. $\bigwedge_{e \in C} H \wedge V = v \Rightarrow [e](V \prec v)$ (generated by Rodin for *convergent* events): convergent events make the variant decrease;
3. $H \wedge V \neq \emptyset \Rightarrow \bigvee_{e \in C} \text{enabled}(e)$ (manually added as a theorem to be proved): some of the convergent events are enabled while the variant is not empty.
4. $H \wedge V = \emptyset \Rightarrow Q$ (manually added as a theorem to be proved): when the variant is empty, the targetted property is satisfied.

where $C \subseteq E$ is a selected set of convergent events and V is a set expression over state variables (both provided by the user). The correctness of these proof obligations heavily relies on strong fairness between two classes of events: enabled convergent events should eventually be fired for the variant to decrease.

A variation of this proof rule may be used when Q is reached before the variant V becomes empty. Obligation (3) is changed as follows:

3a. $H \wedge \neg Q \Rightarrow \bigvee_{e \in C} \text{enabled}(e)$ (manually added as a theorem to be proved): some of the convergent events are enabled while the targetted property is not reached.
3b. $H \wedge Q \Rightarrow \bigwedge_{e \in E} [e](Q)$ (generated by Rodin if Q is declared invariant): Q is stable.

3 Chord Correctness

In order to formalize a problem, the choice of an appropriate mathematical structure is crucial. Indeed, it can ease not only the specification of properties but

[7] Given an event e, $[e](p)$ is the weakest precondition ensuring that e terminates in a state satisfying p.

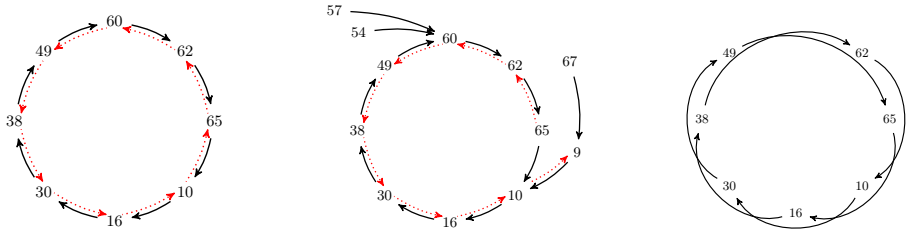

Fig. 1. Some Chord networks: in the ideal state (left), in an arbitrary state (center), loopy (right) (solid edges: `bestSucc`, dotted edges: `prdc`)

also the proof of some of them, in case we can take benefit from meta-properties of the mathematical structure. In our context, an abstract view of a Chord network consists of the total function `bestSucc` over the set MEMBERS of live nodes. As it is the case for every total function over a finite set, its graph is a directed pseudoforest. Thus, the existence of a *ring* of live nodes that is formed by the `bestSucc` relation is directly deduced from the representation of the network through a total function, without any additional hypotheses. Similarly, the fact that all the live nodes are located in the ring(s) is equivalent to `bestSucc` being surjective over MEMBERS. For instance, in the networks on the left-hand side (ideal) and on the right-hand side (loopy) of fig. 1, all the nodes are in the ring: `bestSucc` is surjective. This is not the case for the network in the center. The nodes that are not part of the ring form the *appendages*.

A key notion of safe networks identified by [23] distinguishes between the ideal and the loopy networks. This is the notion of *principal* node, which relates the structure of a network (modeled through `bestSucc` in our case) to the ordering over the node identifier space ($\rightsquigarrow$ in our context). As we will see in the next section, a loopy network does not have any principal node.

In section 3.1 we develop on some results about functions over finite sets, which are not Chord-specific, and in section 3.2 we present the properties of a Chord network.

3.1 Generic Properties

We now present some results about relations and functions over finite sets. These are not Chord-specific, but still useful to prove Chord correctness.

Theorem 1 (Pigeonhole principle). *Given a finite set E, a function $f : E \to E$ is injective if and only if it is surjective.*

A fundamental element in the proof of Chord correctness is the concept of principal nodes, introduced in [23] in the context of a Chord network. We generalize here the definition of a principal node w.r.t. an arbitrary relation r over the identifier space NODE.

Definition 1 (Principal). *Given a binary relation $r \subseteq \text{NODE} \times \text{NODE}$ over the set of nodes, the set* $\texttt{principals}(r)$ *of* principal nodes for r *is the set of nodes that are not skipped by any pair in r:*

$$\texttt{principals}(r) \triangleq \{p \in \text{NODE} \mid \forall \langle n, m \rangle \in r \cdot p \notin n \rightharpoonup m\}$$

The following lemma and theorem will be useful to show that, in the context of a network without appendages, one principal node is enough to ensure that all nodes are correctly located.

Lemma 1. *Given a subset $E \subseteq \text{NODE}$ of nodes and a surjective function $f : E \to E$, if a node p is principal for f, then its next neighbour according to $\rightharpoonup$ in E is also principal for f.*

Proof. Suppose that $\texttt{next}(p)$ is not principal for f. Then, it is between some x and $f(x)$. As p is principal, we have $x = p$. f being surjective, there exists y s.t. $f(y) = \texttt{next}(p)$. As p is principal, $y = p$. Thus $f(p) = \texttt{next}(p) = x$, which contradicts the fact that $\texttt{next}(p)$ is between x and $f(x)$.

Theorem 2 (Principal for a injective (or surjective) total function). *Given a subset $E \subseteq \text{NODE}$ of nodes, and a surjective (or injective) total function $f : E \to E$, if there is some principal node in E for f, then every node in E is principal for f.*

Proof. The proof is straightforward using Lemma 1 and the pigeonhole principle.

3.2 Chord Properties

The authors of Chord have provided explicit properties that ensure correct data delivery [10, 17]. They define in particular the *ideal state* of a network.

Definition 2 (Ideal state). *A Chord network is in an* ideal *state if:*

1. *the first successor and the predecessor of every live node are alive:* $\forall n \in \text{MEMBERS} \cdot n.\texttt{succ}[1] \in \text{MEMBERS} \land n.\texttt{prdc} \in \text{MEMBERS}$
2. *the successor relation* $\texttt{bestSucc}$[8] *forms a single ring of nodes (every live node is in the ring):* $\forall n_1, n_2 \in \text{NODE} \cdot n_2 \in n_1.\texttt{bestSucc}^+ \land n_1 \in n_2.\texttt{bestSucc}^+$, *where* $\texttt{bestSucc}^+$ *is the transitive closure of* $\texttt{bestSucc}$
3. $\texttt{bestSucc}$ *provides the nearest successor of each node according to the identifier order:* $\forall n \in \text{MEMBERS} \cdot n \rightharpoonup n.\texttt{bestSucc} \cap \text{MEMBERS} = \emptyset$
4. $\texttt{prdc}$ *provides the nearest predecessor of each node according to the identifier order:* $\forall n \in \text{MEMBERS} \cdot n.\texttt{prdc} \rightharpoonup n \cap \text{MEMBERS} = \emptyset$
5. *the tail of the successor list of each node is equal to the successor list of its first successor (with the last entry removed):* $\forall n \in \text{MEMBERS} \cdot \forall i \in 2..K \cdot n.\texttt{succ}[i] = n.\texttt{succ}[1].\texttt{succ}[i-1]$

[8] Since all the first successors are alive in the ideal state, $\texttt{bestSucc}$ always points to the first successor.

In the following, we write `ideal` *for the conjunction of the above five properties defining the ideal state.*

As explained in section 2 informally, the ideal state cannot be continuously ensured because nodes can dynamically join and leave the network. The goal of the maintenance protocol is thus to keep the network in a repairable state so that it will be fixed eventually.

Definition 3 (Correctness). *If eventually no node joins or leaves the network anymore, the network will eventually reach the ideal state and remain in it.*

We will prove the convergence of Chord to the ideal state by relying on inductive invariants (section 3.2.1) and on variants (section 4).

3.2.1 Chord Invariants In this section, we exhibit an inductive invariant, which is useful to prove the correctness property. It is inspired by Zave's work [23] and consists of three properties. With respect to this pioneering work, the property related to principal nodes is logically weakened and a technical property, related to our model-specific variables, is added.

Property 1 (`SomeLiveSuccessor`). *A network satisfies* `SomeLiveSuccessor` *if each live node has a live successor:* $\forall n \in \text{MEMBERS} \cdot \exists i \in 1..K \cdot n.\texttt{succ}[i] \in \text{MEMBERS}$.

`SomePrincipal` states that there is some principal among the live nodes. Let us first instantiate the definition of principal, from section 3.1, for a Chord network.

Definition 4 (Chord principal). *A Chord principal is a member that is not "skipped" in any successor list. More formally, a node* $p \in \text{MEMBERS}$ *is a* Chord principal *if, for any node* $n \in \text{MEMBERS}$ *s.t.* $n.\texttt{succ} = [n_1, \ldots, n_K]$, $p \notin n \rightsquigarrow n_1$ *and* $p \notin n_i \rightsquigarrow n_{i+1}$ *for* $i \in 1..(K-1)$.

Proposition 1 (Chord principal). *A node is a chord principal iff it is a member that is a principal for the relation* `hops`*, where :*

$$\begin{aligned}\texttt{hops} \triangleq\ & \{\langle m, m.\texttt{succ}[1]\rangle \mid m \in \text{MEMBERS}\} \\ & \cup \{\langle m.\texttt{succ}[i], m.\texttt{succ}[i+1]\rangle \mid m \in \text{MEMBERS} \text{ and } i \in 1..(K-1)\}\end{aligned}$$

Considering the relation `bestSucc` instead of the relation `hops`, *i.e.*, having a more abstract view of the successor relation, we have the following proposition:

Proposition 2 (Principal for `bestSucc`). *Given a Chord network, if a node* n *is a Chord principal,* i.e.*, a member that is a principal for* `hops`*, then it is a principal node for* `bestSucc`*:* $\texttt{principals}(\texttt{hops}) \subseteq \texttt{principals}(\texttt{bestSucc})$.

We can now state the property `SomePrincipal`.

Property 2 (`SomePrincipal`). *A Chord network satisfies* `SomePrincipal` *if there is some live node which is a Chord principal:* $\texttt{principals}(\texttt{hops}) \cap \text{MEMBERS} \neq \emptyset$.

Notice this is logically weaker than the property from [23], where the number of principals was required to be greater than the size of successor lists, as discussed above.

The following property is related to the model-specific variable `Stabilizing`, which records the fact that a node n has to take a node m as its future successor.

Property 3 (`StabBetterThanSucc`). *A Chord network satisfies* `StabBetterThanSucc` *if for every live node n having a pending stabilization, the candidate for stabilization is better than the current successor of n:*

$$\forall n : \text{MEMBERS} \cdot n \in \mathtt{dom}(\mathtt{Stabilizing}) \Rightarrow \mathtt{Stabilizing}(n) \in n \rightsquigarrow n.\mathtt{succ}[1]$$ [9]

Theorem 3 (Inductive invariant). *The following property is preserved by all of the operations of the Chord protocol, except* `fail`*:*

$$\mathtt{SomeLiveSuccessor} \wedge \mathtt{SomePrincipal} \wedge \mathtt{StabBetterThanSucc}$$

The proof of this theorem is mechanized with Rodin.

Operating Assumptions. *Our proof of correctness for Chord relies on the critical operating assumptions that no failure "breaks" the invariant*[10]*.:*

1. *No failure leaves a node without live node in its successor list.*
2. *No failure leaves the network without any principal node.*

The assumption (1), saying that each successor list always includes a live node, was present in the original Chord article [17]. Indeed, having a list of successors prevents from the failure of a successor as soon as there are other nodes left in the successor list. The assumption (2) comes from Chord property `SomePrincipal`, which is an adaptation from the invariant property exhibited in [23], where the author explained that when a node joins the network, it becomes a principal as soon as its K preceding nodes are aware of its presence. Assuming the existence of a minimal number of principal nodes ($K + 1$ in [23]) is then reasonable, especially as we assume the existence of only one principal node in this article.

Notice that we also relaxed the assumptions from [23] about the minimal size of the network and about the absence of duplication in successor lists.

3.2.2 Always-True Properties We now define important structural properties and show that they are actually implied by the inductive invariant.

Property 4 (`AtMostOneRing`). *A Chord network satisfies the property* `AtMostOneRing` *if any two ring members can access each other through* $\mathtt{bestSucc}^+$*.*

$$\forall n_1, n_2 \in \text{MEMBERS} \cdot (n_1 \in n_1.\mathtt{bestSucc}^+ \wedge n_2 \in n_2.\mathtt{bestSucc}^+) \Rightarrow (n_1 \in n_2.\mathtt{bestSucc}^+ \wedge n_2 \in n_1.\mathtt{bestSucc}^+)$$

[9] `dom` denotes the domain of a relation or a function.

[10] The Chord property 3 about the `Stabilizing` function is preserved by `fail` and thus does not impact operating assumptions.

Theorem 4. *Given a Chord network,* `SomePrincipal` *implies* `AtMostOneRing`.

Proof (sketch). Suppose that `SomePrincipal` is true. Then, there is a Chord principal, which is also a principal node for `bestSucc` (from Property 2). Also suppose that `AtMostOneRing` is false. There there are two nodes n_1 and n_2 that are in two unconnected `bestSucc`-"rings". Considering the first ring, any node outside this ring is necessary "skipped" by `bestSucc`: all the principal nodes are thus in the first ring. Similarly, we can conclude that all the principal nodes are in the second ring. Contradiction. □

Property 5. *A Chord network satisfies* `DistinctFirstSuccs` *if the successor lists include no duplicated node up to the first live node:*

$$\forall n \in \text{NODE} \cdot \forall j \leqslant \mathit{fl}_n \cdot \forall i \in 1..j-1 \cdot n.\texttt{succ}[i] \neq n.\texttt{succ}[j]$$

where fl_n is the index of the first live node in $n.$`succ`.

Theorem 5. `SomePrincipal` *implies* `DistinctFirstSuccs`.

Proof (sketch). Suppose that `SomePrincipal` is true and `DistinctFirstSuccs` is false. Then, there is a node n s.t. in $n.$`succ`, there is a duplicated node n' before the first live node in $n.$`succ`. Since every node except n' is in the set $n' \rightsquigarrow n'$, there cannot be a Chord principal different from n', which contradicts `SomePrincipal`, because n' is not a member. □

Property 6. *A Chord network satisfies* `OrderedFirstSuccs` *if the successor lists are ordered according to* $\rightsquigarrow$ *up to the first live node in the list:*

$$\forall n \in \text{NODE} \cdot \forall j \leqslant \mathit{fl}_n \cdot \forall i \in 1..j-1 \cdot n.\texttt{succ}[i] \in n \rightsquigarrow n.\texttt{succ}[j]$$

where fl_n is the index of the first live node in $n.$`succ`.

Theorem 6. *Given a Chord network,* `SomePrincipal` *implies* `OrderedFirstSuccs`.

Proof (sketch). Suppose that `SomePrincipal` is true and `OrderedFirstSuccs` is false. Then, there are n, i, j as in the theorem statement s.t. $n.\texttt{succ}[i] \notin n \rightsquigarrow n.\texttt{succ}[j]$. Since these three nodes are distinct (from theorem 5), we have $n.\texttt{succ}[j] \in n \rightsquigarrow n.\texttt{succ}[i]$. Then, the properties of $\rightsquigarrow$ imply that every node except $n.\texttt{succ}[i]$ is included in $n \rightsquigarrow n.\texttt{succ}[i] \cup n.\texttt{succ}[i] \rightsquigarrow n.\texttt{succ}[j]$. From `SomePrincipal`, there is a live node p which is principal. Since p is a live node, it is distinct from $n.\texttt{succ}[i]$. It is then skipped by some pair in $n.$`succ`, which leads to a contradiction. □

4 Phase-based Convergence Proof

We now show that in the absence of `join` and `fail` events, the system eventually reaches the *ideal* state and remains in it. To do so, we introduce four intermediate macro-states, which are stable[11] under the considered hypothesis and reached successively:

[11] Once a macro-state is reached, the system cannot leave it.

MS1. `Rectifying` and `prdc` in MEMBERS.

MS2. the first successor is a member: $n.\texttt{succ}[1] \in \text{MEMBERS}$.

MS3. `Stabilizing` only includes members: $\texttt{ran}(\texttt{Stabilizing}) \subseteq \text{MEMBERS}$[12].

MS4. `prdc` is the inverse of `bestSucc` and both `Stabilizing` and `Rectifying` are empty for members:
$\forall n \in \text{MEMBERS} \cdot n.\texttt{Stabilizing} = \emptyset \wedge n.\texttt{Rectifying} = \emptyset \wedge$
$n.\texttt{bestSucc}.\texttt{prdc} = n$

MS5. the tail of the successor list of each node is equal to the successor list of its first successor (with the last entry removed):
$\forall n \in \text{MEMBERS} \cdot \forall i \in 2..K \cdot n.\texttt{succ}[i] = n.\texttt{succ}[1].\texttt{succ}[i-1]$

Ideal. We then prove that MS5 implies that the network is ideal.

This phase-based proof allows us to avoid a monolithic convergence proof which would require finding a complex *variant*. Each phase (and sub-phase) relies on a small variant, except the fourth phase (reaching MS4). It relies on the proof method presented in section 2.3 and thus on fairness hypotheses. The proof was mechanized in Rodin from our Event-B model, where the guards of `join` and `fail` were set to `false`[13].

4.1 Reaching MS1: `Rectifying` and `prdc` in members

This phase is split into two steps: reaching MS1a from a state satisfying the inductive invariant, and reaching MS1b from MS1a.

MS1a. $\texttt{ran}(\texttt{Rectifying}) \subseteq \text{MEMBERS}$
We split the event `rectify` in two events, one guarded by $\texttt{newPrdc} \notin \texttt{members}$ and the other by $\texttt{newPrdc} \in \texttt{members}$. The variant is the set $\texttt{Rectifying} \setminus \text{NODE} \times \text{MEMBERS}$. The event that is guarded by the negative membership condition makes the variant decrease (it is tagged *convergent* in Event-B) while the other events do not make it increase (*anticipated* in Event-B). As long as `Rectifying` includes non members, the convergent event is enabled. So, under the fairness hypothesis, MS1a will be reached eventually.

MS1b. $\texttt{prdc} \in \text{MEMBERS} \nrightarrow \text{MEMBERS}$
This property is shown by introducing the variant $\texttt{prdc}^{-1}[\text{NODE} \setminus \text{MEMBERS}]$. The `rectifyNull` event decreases the variant (and other events do not increase it). It is enabled as long as the variant is not empty, which ensures the convergence from MS1a to MS1b.

[12] `ran` denotes the range of a relation or a function.

[13] Technically, we have an Event-B model for each phase defined as a refinement of the Event-B machine modelling the Chord protocol, where the MS of the preceding phase is stated as an invariant of the current phase.

4.2 Reaching MS2: the first successor is a member

$$\forall n \in \text{MEMBERS} \cdot n.\texttt{succ}[1] \in \text{MEMBERS}$$

Notice that this is equivalent to $\forall n \in \text{MEMBERS} \cdot n.\texttt{bestSucc} = n.\texttt{succ}[1]$. In order to ease the reasoning, given a member node n, we call $\texttt{zombies}(n)$ the set of non member nodes preceding $n.\texttt{bestSucc}$ in its successor list. So, the objective of this phase is to reach a state where the zombie sets are empty. It is split into two steps: reaching MS2a from MS1, and reaching MS2b from MS2a.

MS2a. each node in the stabilizing state has no zombie successors:
$\forall n \in \texttt{dom}(\texttt{Stabilizing}) \cdot \texttt{zombies}(n) = \emptyset$.
The event `stabilizeFromFstPrdc` is split to introduce the guard `zombies(m)` $= \emptyset$ and its negation. The event with the negative guard makes the variant `dom (Stabilizing)` $\setminus$ `{m` $\mid$ `m`$\in$MEMBERS $\wedge$ `zombies(m)` $= \emptyset$`}` decrease while others do not increase it.

MS2b. each member has no zombie successors: $\forall n \in \text{MEMBERS} \cdot \texttt{zombies}(n) = \emptyset$
This property is ensured thanks to the event `stabilizeFromFst@1` which removes one element from a non empty zombie set of a member node that has no `Stabilizing` memory context. In MS1a, this condition is true. For this phase the variant is $\{\langle m, b\rangle \cdot m \in \text{MEMBERS} \wedge b \in \texttt{zombies}(m)\}$.

4.3 Reaching MS3: `Stabilizing` only includes members

$$\texttt{ran}(\texttt{Stabilizing}) \subseteq \text{MEMBERS}$$

We take as variant the set of the pairs (n_1, n_2) in `Stabilizing` s.t. n_2 is not a member. The sub-event `stabilizeFromFstPrdc@1` makes the variant $\texttt{Stabilizing} \cap \text{MEMBERS} \times (\text{NODE} \setminus \text{MEMBERS})$ decrease. It is enabled while the variant is not empty.

4.4 Reaching MS4: `prdc` is the inverse of `bestSucc` and the `Rectifying` and `Stabilizing` sets of each node are empty

$$\forall n \in \text{MEMBERS} \cdot n.\texttt{Stabilizing} = \emptyset \wedge n.\texttt{Rectifying} = \emptyset \wedge n.\texttt{bestSucc}.\texttt{prdc} = n$$

This property is proved by introducing a complex variant which is the combination of four sets of node pairs. The events of the protocol make the variant decrease by moving some pairs from one of the sets to another one of lower weight, or by removing some pair from the lowest set. Other transitions let the sets unchanged. The following sets are the following in decreasing order of importance (for the variant)[14]:

1. $\{\langle x, y\rangle \mid x \in \text{MEMBERS} \wedge y \in \text{MEMBERS} \wedge y \in x \rightarrowtail \texttt{bestSucc}(x))\} \setminus \texttt{Stabilizing}$

[14] In Event-B, this structured variant is encoded as a single set using the Cartesian product and union.

2. $\{\langle x, y\rangle \mid x \in \text{MEMBERS} \wedge y \in \text{MEMBERS} \wedge (y \in \texttt{dom}(\texttt{prdc}) \wedge \texttt{prdc}(y) \in \text{MEMBERS} \Rightarrow x \in \texttt{prdc}(y) \leadsto y)\} \setminus \texttt{Rectifying}^{-1}$
3. $\texttt{Stabilizing} \cap \texttt{Rectifying}^{-1}$
4. $\texttt{Stabilizing}$
5. $\texttt{Rectifying}^{-1}$

The following events make the variant decrease: `stabilizeFromFst@{1,2sta}`, `stabilizeFromFstPrdc@2`, `rectify`. Besides, the event `stabilizeFromFst@2rct` must be split to introduce the guard $\langle \texttt{m.suc}, \texttt{m}\rangle \notin \texttt{Rectifying}$. The sub-event having this guard true makes also the variant decrease. The other events do not increase it.

4.5 Reaching MS5: the tail of the successor list of each node is equal to the successor list of its first successor

$$\forall n \in \text{MEMBERS} \cdot \forall i \in 2..K \cdot n.\texttt{succ}[i] = n.\texttt{succ}[1].\texttt{succ}[i-1]$$

In this step, we need to manage the concrete successor list of each member node while its abstraction with `bestSucc` and a zombie set was sufficient to verify the previous phases. Data refinement is used to replace these two variables by a unique successor list. Verifying its correctness is not automatic as automation is weaker with lists: numerous user-provided case splitting and quantifier instantiations are required.

Then, in order to prove the convergence to MS5, an auxiliary variable E is introduced: it includes the pairs (m, i) such that the successor list of m is correct up to position i. More precisely, E is introduced with the following invariant properties:

$$E \subseteq \text{MEMBERS} \times 1..K \qquad \text{MEMBERS} \times \{1\} \subseteq E$$
$$\forall m, i \cdot \langle m, i\rangle \in E \wedge i > 1 \Rightarrow m.\texttt{succ}[1].\texttt{succ}[i-1] \in E$$
$$\forall m, i \cdot \langle m, i\rangle \in E \wedge i > 1 \Rightarrow \{m\} \times 1..i \subseteq E$$
$$\forall m, i \cdot \langle m, i\rangle \in E \wedge i > 1 \Rightarrow m.\texttt{succ}[i] = m.\texttt{succ}[1].\texttt{succ}[i-1]$$

Thanks to MS4, we can start with $\text{MEMBERS} \times \{1\}$. Then, thanks to fairness, the event `stabilizeFromFst@2rct` which copies the successor list of one node to its predecessor will eventually saturate E. This property is ensured by taking $(\text{MEMBERS} \times 1..K) \setminus E$ as variant and by splitting the selected event s.t. it ensures progress.

4.6 Reaching the ideal state

By using the results of section 3, we show that the properties of MS5 imply that the network is in the ideal state. Indeed, in MS4, `bestSucc` is necessarily injective. From the invariant, we have that there is at least one principal node. By theorem 2 and proposition 2, all nodes are principal for `bestSucc`, which means that no node is skipped by `bestSucc`. Moreover the last property defining the ideal state exactly matches the definition of MS5. The five properties of an ideal state are thus fulfilled.

5 Related Work

Chord is a popular protocol but also, since the seminal work of Zave [20], a popular test-bed for formal verification. However, most work [3, 9, 15, 20–22] has focused on proving *safety*, sometimes with manual proofs only, while the correctness property for Chord maintenance, addressed here, is a liveness property. Zave [23] carried out a manual proof of liveness and discovered the fundamental notion of *principal*. Some of the authors of the present paper analyzed liveness in an automated way using Electrum [5] but for small networks. To the best of our knowledge, this work is the first to address the liveness property of Chord in a parameterized setting and using a much automated, mechanical proof.

Other distributed system protocols have been formally studied using "high-level" specification languages. For instance, Pastry was analyzed using TLA$^+$ [13]; similar work used Event-B [16] or ASM [12] to partly verify other protocols. However, these studies are limited to the verification of safety properties.

Verdi [19] and IronFleet [6] address the question of provably-correct *implementations* of distributed protocols while our approach is markedly at a more abstract level, in particular to favor proof automation. Our work is also focused on a stabilization property for which we developed a specific proof method. Finally, proof automation for liveness of parameterized or even arbitrary infinite-state distributed systems is the subject of recent work such as Ivy [14] but, as far as we know, a fair amount of manual intervention is still needed.

6 Conclusion

In this article, we proposed a mechanized correctness proof of the Chord maintenance protocol. We address a particular form of liveness property (stabilization) over a network of arbitrary size. On the logical side, we weakened the operating assumption related to principal nodes stated in [23], as well as the one requiring a minimal number of nodes in the network. However, the practical consequences of this weakening remain to be assessed quantitatively.

As future work, we intend to develop some automated support to stabilization proofs following the method exhibited in section 4. Another line of work is to refine our model with less abstract types (*e.g.* FIFO for asynchronous communication). Both directions could contribute to the design of a framework for (1) modelling knowledge in distributed systems, and (2) supporting liveness proofs under fairness assumptions, with an important degree of automation.

Acknowledgements. We warmly thank Pamela Zave for insightful discussions on the protocol and for her thorough reading of this article.

J. Brunel and D. Chemouil were partly financed by the European Regional Development Fund (ERDF) through the Operational Programme for Competitiveness and Internationalisation (COMPETE2020) and by National Funds through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT) within project POCI-01-0145-FEDER-016826; and within the French Research Agency project FORMEDICIS (ANR-16-CE25-0007).

References

1. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2009). https://doi.org/10.1017/cbo9781139195881
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer **12**(6), 447–466 (Nov 2010). https://doi.org/10.1007/s10009-010-0145-y
3. Bakhshi, R., Gurov, D.: Verification of peer-to-peer algorithms: A case study. Electronic Notes in Theoretical Computer Science **181**, 35–47 (2007). https://doi.org/10.1016/j.entcs.2007.01.052
4. Bodeveix, J.P., Brunel, J., Chemouil, D., Filali, M.: A model in Event-B of the Chord protocol (Jul 2019). https://doi.org/10.5281/zenodo.3271455
5. Brunel, J., Chemouil, D., Tawa, J.: Analyzing the Fundamental Liveness Property of the Chord Protocol. In: Formal Methods in Computer-Aided Design. Austin (USA) (Oct 2018). https://doi.org/10.23919/fmcad.2018.8603001, https://hal.archives-ouvertes.fr/hal-01862755
6. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: Proving practical distributed systems correct. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP). ACM – Association for Computing Machinery (October 2015). https://doi.org/10.1145/2815400.2815428
7. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
8. Lamport, L.: Specifying systems: the TLA$^+$ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc. (2002)
9. Li, X., Misra, J., Plaxton, C.G.: Active and concurrent topology maintenance. In: International Symposium on Distributed Computing. pp. 320–334. Springer (2004). https://doi.org/10.1007/978-3-540-30186-8_23
10. Liben-Nowell, D., Balakrishnan, H., Karger, D.: Analysis of the evolution of peer-to-peer systems. In: Proceedings of the twenty-first annual symposium on Principles of distributed computing. pp. 233–242. ACM (2002). https://doi.org/10.1145/571860.571863
11. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In: Foundations of Software Engineering (2016). https://doi.org/10.1145/2950290.2950318
12. Marinković, B., Glavan, P., Ognjanović, Z.: Proving properties of the chord protocol using the asm formalism. Theoretical Computer Science **756**, 64 – 93 (2019). https://doi.org/https://doi.org/10.1016/j.tcs.2018.10.025, http://www.sciencedirect.com/science/article/pii/S0304397518306467
13. Merz, S., Lu, T., Weidenbach, C.: Towards Verification of the Pastry Protocol using TLA$^+$. In: 31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems. vol. 6722 (2011). https://doi.org/10.1007/978-3-642-21461-5_16
14. Padon, O., Hoenicke, J., Losa, G., Podelski, A., Sagiv, M., Shoham, S.: Reducing liveness to safety in first-order logic. PACMPL **2**(POPL), 26:1–26:33 (2018). https://doi.org/10.1145/3158114
15. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 614–630 (2016). https://doi.org/10.1145/2908080.2908118

16. Risson, J., Robinson, K., Moors, T.: Fault tolerant active rings for structured peer-to-peer overlays. In: Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on. pp. 18–25. IEEE (2005). https://doi.org/10.1109/lcn.2005.69
17. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review **31**(4), 149–160 (2001). https://doi.org/10.1145/964723.383071
18. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Transactions on Networking (TON) **11**(1), 17–32 (2003). https://doi.org/10.1109/tnet.2002.808407
19. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: 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. pp. 357–368 (2015). https://doi.org/10.1145/2737924.2737958
20. Zave, P.: Why the Chord ring-maintenance protocol is not correct. Tech. rep., AT&T Research, Tech. Rep (2011)
21. Zave, P.: Using lightweight modeling to understand Chord. ACM SIGCOMM Computer Communication Review **42**(2), 49–57 (2012). https://doi.org/10.1145/2185376.2185383
22. Zave, P.: A practical comparison of Alloy and SPIN. Formal Aspects of Computing **27**(2), 239 (2015). https://doi.org/10.1007/s00165-014-0302-2
23. Zave, P.: Reasoning about identifier spaces: How to make Chord correct. IEEE Transactions on Software Engineering **43**(12), 1144–1156 (Dec 2017). https://doi.org/10.1109/TSE.2017.2655056