Analyzing the Fundamental Liveness Property of the Chord Protocol

Julien Brunel
ONERA DTIS & Univ. Toulouse
F-31055 Toulouse, France
julien.brunel@onera.fr

David Chemouil
ONERA DTIS & Univ. Toulouse
F-31055 Toulouse, France
david.chemouil@onera.fr

Jeanne Tawa
ONERA DTIS & Univ. Toulouse
F-31055 Toulouse, France
jeanne.tawa@onera.fr

***Abstract*—Chord is a protocol that provides a scalable distributed hash table over an underlying peer-to-peer network. Since it combines data structures, asynchronous communications, concurrency, and fault tolerance, it features rich structural and temporal properties that make it an interesting target for formal specification and verification. Previous work has mainly focused on automatic proofs of safety properties or manual proofs of the full correctness of the protocol (a liveness property). In this paper, we report on analyzing automatically the correctness of Chord with the Electrum language (developed in former work) on small instance of networks. In particular, we were able to find various corner cases in previous work and showed that the protocol was not correct as described there. We fixed all these issues and provided a version of protocol for which we were not able to find any counterexample using our method.**

***Index Terms*—Chord protocol, distributed systems, formal specification and verification, Electrum**

I. Introduction

Peer-to-peer systems are distributed systems without hierarchical organization or centralized control. They are an alternative to the traditional client-server model and enjoy interesting properties in terms of scalability, robustness and cost. Chord [1]–[3] is a one of the most popular peer-to-peer systems. It is a protocol and algorithm for a peer-to-peer distributed hash table (DHT). A DHT stores key-value pairs by assigning keys to different nodes (basically computers) in the network. Chord addresses the efficient and robust localization of data in such a network. When Chord was initially presented, three main qualities were highlighted: its simplicity, its provable performance and its provable correctness. Although the first two claims are true, proving the Chord correctness turns out to be a hard task, as showed by numerous works by P. Zave [4]–[8].

In Chord, each node has an identifier and can reach other nodes using pointers to other identifiers. The nodes and their pointers form a topology which is essential to ensure the correct localization of data in the network. Because of the fact autonomous nodes may join or leave the network (or fail) at any time, the topology is always evolving. A key aspect of the Chord protocol consists in the definition of maintenance operations that are in charge of repairing the network topology so that the data stored in any node keeps being reachable from any other node, despite failures, joins and departures.

Thus, the correctness of Chord deals with the network topology. In fact, the nodes and their successor pointers have to form a ring, so that each node is accessible from any other node. Since nodes can join and leave the network, the ring topology cannot always be ensured. That is why the the correctness property of Chord is expressed as follows: *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 keep it*. So, the correctness of Chord is not only about the structure of the system, but also about its temporal evolution: it is in fact a liveness property. This twofold nature is one of the reasons for the hardness to prove Chord correctness.

We recently developed Electrum [9], a specification language based on First-Order Linear Temporal Logic, with which both structural and temporal properties can easily be defined and checked. The Electrum language is inspired by Alloy [10] for its structural concepts and by Linear Temporal Logic [11] for its temporal concepts.

In this article, we propose a formal description of the Chord protocol in Electrum and focus on proving its correctness. We show the following benefits of our approach:

- the Electrum ability to deal with structural aspects makes the specification of the network topology straightforward;
- the Electrum ability to deal with temporal aspects fits with the specification of the network evolution (throughout the execution of the maintenance operations) and makes the specification of the correctness property, which is a liveness property, direct;
- the automatic verification of the full correctness property is performed for the first time (only for a limited number of nodes though)
- thanks to the quick feedback to the user, we have been able to the detect several shortcomings and corner cases in the previous formalization of the protocol, and to clearly identify temporal hypotheses on the ordering of the maintenance operations (fairness properties) that are necessary to ensure the correctness.

The rest of this paper is structured as follows. In Sect. II we briefly present the Chord protocol. In sect. III, we give an overview of Electrum, and formalize Chord in sect. IV. In Sect. V, we evaluate the formal verification of our Chord

model. In Sect. VI, we highlight important aspects of our study and compare to related work. We then conclude in Sect. VII.

II. The Chord Protocol

Chord is a distributed lookup protocol which addresses an essential issue of peer-to-peer applications: the efficient localization of the network node that stores the desired data. An important quality that probably explains the popularity of Chord is its simplicity. Indeed, Chord makes no use of synchronization or timing constraints on distributed nodes, and each atomic operation involves a single node. As claimed by the authors, this simplicity makes Chord easy to implement and extend. Other interesting features of Chord are its provable performance and its scalability. However, contrary to another claim, proving the correctness of Chord. *i.e.*, the reachability of the data, is not an easy task.

A. The Network Structure

In a Chord network, each node has an identifier (the m-bit hash of its IP address). Pairs of keys and associated data are stored in nodes. Every node has a *successor list* of pointers to other nodes. We refer to the first element of this list as the *successor*. The goal of having a list of successors instead of a single one is to be robust to the failures: if a node leaves the network, its predecessor still has successors in the network. Besides, each node also has a pointer to its *predecessor*. This is useful in the execution of the Chord maintenance operations.

When a network is structured as a ring according to the relation induced by the *successor* pointers and when the order of identifiers complies with the order of the *successor* pointers, then each node is accessible from any other node, *i.e.* any data is accessible from any node. We say that such a network is in an *ideal* state.

Since nodes can join and leave the network at any time, the ring structure cannot be continuously ensured. For instance, nodes joining a ring create an appendage. The maintenance operations aim to recover a ring structure eventually, despite the fact nodes join and leave the network.

B. Network Properties

The authors of Chord have provided explicit properties of the network that ensure correct data delivery [2]. They define in particular the *ideal state* of a network, which we have introduced informally in the previous section, and a temporary imperfect state, which we call a *valid state* following [4]. As our study only deals with the correctness of the protocol, we do not present the quantitative and probabilistic properties mentioned in the original Chord articles.

Let us first present some notations and preliminary definitions. In the following, we will denote the successor (resp. predecessor) of a node n by n.SUCCESSOR (resp. n.PREDECESSOR). A Chord network is *locally consistent* if, for any node n, we have $(n.\text{SUCCESSOR}).\text{PREDECESSOR} = n$. A Chord network is *globally consistent* if, for each node n_1, there is no node n_2 in the same ring as n_1 such that $n_1 < n_2 < n_1.\text{SUCCESSOR}$. A Chord network is *loopy* if it is locally consistent but globally inconsistent.

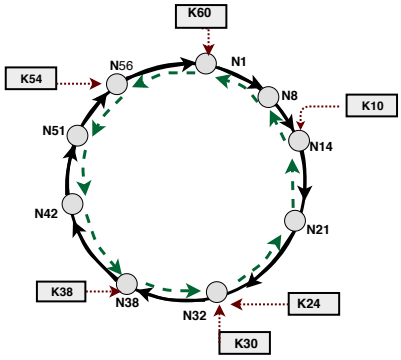

Fig. 1. A Chord network in an ideal state (successor pointers are shown as a bold arrows, predecessor pointers as dashed arrows and key/node mappings as dotted arrows).

Definition 1: A Chord network is in an *ideal* state if:

- *ring:* the successor relation forms a single ring of nodes (every node is in the ring);
- *non-loopiness:* the ring is locally and globally consistent;
- *successor list validity:* the successor list (of size k) of each node n contains the first k nodes that follow n in the ring.

Fig. 1 shows a Chord network in an ideal state, with nine nodes and storing six key-data pairs (we only represent the keys). Each key is stored in the node with the least identifier among the nodes having a greater identifier than the key. For example, key K10 is stored in node N14.

As explained above, joins and fails of nodes force the network in a non-ideal state. But the maintenance operations of Chord aim at recovering from such non-ideal states.

In order to characterize these non-ideal states, we introduce the notion of *valid* states (following [2] and [4]) which allow some nodes not to be in the ring, but in *appendages* of the ring. For a node n in the ring, there may be a tree of nodes rooted at n, consisting of nodes that have recently joined the network and are not yet in the ring. We refer to this tree as n's appendage and denote it A_n.

Definition 2: A Chord network is in a *valid* state if:

- *connectivity:* a subset of nodes form a ring following the successor relation (there is only one such ring), the rest of the nodes are part of appendages, which are connected to the ring;
- *non-loopiness:*
  - the ring is non-loopy;
  - and for every node n' in an appendage A_n, the path of successors from n' to n is increasing (in the sense of the identifier order).
- *successor list validity:*
  - if n is in the ring, then n.SUCCESSOR is the first live ring node following n (according to the identifier order);

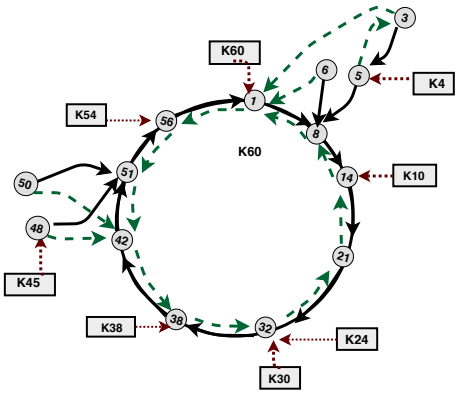

Fig. 2. A Chord network in a valid state.

- if n' is in appendage A_n, then n is the first live ring node following n' (according to the identifier order);
- if the successor list of n.SUCCESSOR skips over a live node n', then n' is not in n successor list.

Fig. 2 shows a Chord network in a validstate.

From these two definitions, the correctness of the Chord protocol can be expressed as follows.

Correctness of the Chord protocol: Starting from a network that is initially valid, in any execution state, if there are no subsequent join or fail events, then the network will eventually become ideal and remain ideal.

C. Chord Events

The operations of the Chord protocol consist of four events (`join`, `fail`, `stabilize` and `rectify`) each of which changes the state of at most one node. A `join` operation occurs when a node joins the network. We then refer to this node as a *member*, or a *live* node. When a node joins the network, it contacts a network member and takes the successor list of this member as its own successor list. It also considers this member as its predecessor. Diagrams (a) and (b) in Fig. 3 show successor and predecessor pointers in a network where node 6 joins by contacting node 8.

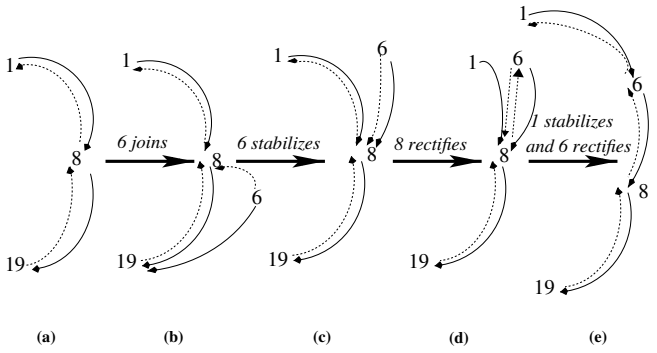

Fig. 3. Chord Events: predecessor pointers are shown as dashed arrows and successor pointers as solid arrows

A `join` event may break the ideal property of the network. In order to recover, every node performs a `stabilize` operation periodically. When a member *stabilizes*, it contacts its successor and asks it about its predecessor identifier. If the predecessor identifier is a better candidate for being its successor than its current successor (according to the identifier order) then it takes this predecessor as its new successor. In diagram (c) of Fig. 3, node 6 stabilizes and takes node 8 as its new successor. The contact a node establishes with its successor during stabilization is also an opportunity to update its full successor list with information from its successor.

After stabilization, the stabilized node *notifies* its successor about its identity. The notified member then executes a `rectify` operation. A notified member must adopt the notifying member as its new predecessor if the notifying member is closer to itself than its current predecessor, or if its current predecessor is dead (see below). Diagram (d) in Fig. 3 shows a `rectify` operation made by node 8 after the notification by node 6.

Finally, a node can leave the network in case of a `fail` operation. Such a node is no longer a member and is referred to as a *dead* node. It obviously does not inform any other node about its departure and still appears in the successor list of other nodes.

a) Operating assumptions: Chord relies on an important assumption, which states that each member always has at least one live successor. In practice, this depends on the size of the successor list, and on the ratio between the occurrence of maintenance operations and the occurrence of failures. For instance, let us suppose that a given live node n has a successor list of size 3, and that all three successors of n fail before any stabilisation and rectification occur, then the network is no longer in a valid state (the ring structure is broken) and the protocol is not able to recover from such a situation.

Another assumption is the perfect communication between a node and its successor, in the sense that each node necessarily answers a query in bounded time. This allows for perfect detection of failures (a successor that does not answer a query before a deadline is considered dead).

III. Electrum in a Nutshell

Electrum [9] is a dynamic extension of Alloy [10] based upon Linear Temporal Logic (LTL). It preserves the flexibility of Alloy while easing the specification of behavioral properties and enabling verification on traces with a bounded or an unbounded number of states.

In Electrum, as in classic class-based modelling, structure is introduced through the declaration of *signatures*, which denote sets of indivisible, immutable and uninterpreted *atoms*; and *fields* (between signatures) that denote flat n-ary relations between sets. Signatures and fields may be constrained by simple *multiplicity* constraints.

Unlike in Alloy however, fields and signatures may either be declared as *static* (by default) or *variable*: the former hold the same valuation throughout a given time trace, while the latter are mutable and hence may see their valuation change at every step of a trace.

If needed, more constraints may be imposed on a specification as *facts*, which are just axioms (*i.e.* statements that every instance of the specification conforms to).

Constraints (formulas) are expressed in a logic comprising both connectives (and quantifiers) of First-Order Logic (FOL) and LTL, with relational expressions as a term language. The latter are built by composing signatures and fields with common set-theoretic operators and relational operators such as the join `.` of two relations or the transitive closure `^` of a relation. Moreover, every relational expression may be primed, referring thus to its valuation in the succeeding state. For ease of specification, parameterized, named expressions and constraints may also be introduced as functions and (resp.) predicates.

Analysis instructions consist of **`run`** and **`check`** commands restricted by *scopes* that determine the maximum number of atoms that will be considered for every signature. A **`run`** instructs the Analyzer to search for an instance (a model, in the logical sense) satisfying a given predicate; while a **`check`** instructs the Analyzer to prove a given assertion (introduced with the **`assert`** keyword) valid *in the given scope*.

Electrum Analyzer[1], an extension of Alloy Analyzer, offers two alternative model-checking techniques: the first implements *bounded model-checking* (BMC) [12], [13] over Alloy itself, thus bounding the number of states in a trace (this is expressed with a bound over a fake **`Time`** signature). The second one relies on the compilation to the NuSMV [14] and nuXmv [15] model-checkers, relying on *unbounded* model-checking (UMC) algorithms.

IV. Formalization of the Protocol

We now present the main aspects of the formalization of the Chord protocol[2], taking inspiration in both the presentation of Chord in [2], referred to as PODC in the rest of this section, and in P. Zave's recent work [4].

A. Data Structures

The main concept in our model is that of a *node*, which corresponds to a Chord node identifier (we conflate Chord nodes, their IP address, and their identifier). As explained before, node (identifiers) are ordered totally.

Recall that a node also maintain a list of successors: its purpose is to recover from failures, and its length defines a threshold for fault tolerance for Chord. To ensure that each node always remains connected to the network after a failure, the minimum length of this list is 2. For the sake of readability, we only show a model with successor list of size 2. Actually, we "unfold" this list and represent it as the datum of two fields `fst` ("first") and `snd` ("second"). We made this choice because using lists here would make the use of *explicit* quantification over all possible lists necessary, a fact that is easily overlooked and that, more importantly, is costly in terms of space.

Finally, to ensure maintenance operations, each node also holds a pointer `prdc` to its predecessor in the network. These three fields may mutate, depending on various events happening in the network, hence they are marked as **`var`**iable. Technically, each of this field denotes a partial function from nodes to nodes, which is specified using the **`lone`** multiplicity (meaning "0 or 1"):

```
open util/ordering[Node] // total ordering on nodes
sig Node {
  var fst, snd, prdc: lone Node,
  var todo: Status → Node }
```

The `todo` field, also present in the declaration, represents *pending operations* that the node will have to perform over another node (hence this field denotes a ternary relation): its use will be detailed later. There are two kinds of such operations, described by a so-called *status* (its formalization is a way of saying that it is an enumeration):

```
abstract sig Status {}
one sig Stabilizing, Rectifying extends Status {}
```

A node is a *member* of the Chord network if its successor pointers effectively point to some nodes (*i.e.* the pointers are not null). This is neatly expressed by introducing a *variable subset signature* that takes its elements among nodes but the valuation of which may change at every instant[3]:

```
var sig members in Node {}
fact membersDef {
  always members = {n: Node | some n.fst && some n.snd}}
```

Now, at every instant, the successor of a node is the first living node among its successors. We specify this as a partial function `succ` which states that the successor of a node is its `fst` field if this field is a member, and its `snd` field otherwise.

```
fun succ: Node → lone Node {
  { m1, m2: members | m1.fst in members ⇒ m2 = m1.fst
                      else m2 = m1.snd } }
```

Using this definition, we can define *ring members* as members belonging to the cycle maintained by Chord. This is once again expressed as a variable subset signature, the elements of which are those that can all reach themselves through the *transitive closure* (`^`) of `succ`:

```
var sig ring in members {}
fact ringDef {
  always ring = { m : members | m in m.^succ } }
```

Finally, the set of *appendages* can simply be defined as those members that are not ring members:

```
fun appendages: set Node { members − ring }
```

B. Network Properties

As nodes are arranged into a cyclic network, their ordering must take into account the fact that the successor of the largest node identifier is the smallest one. Besides, we will often need to compare nodes by checking whether one node is between two others. This is reflected by the following definitions:

[1] Cf. https://haslab.github.io/Electrum.

[2] The full model is available at https://doi.org/10.5281/zenodo.1322052.

[3] **`always`** is the classic G (or □) connective of LTL; **`some`** applied to an expression means "not null"; and `.` is the relational join akin, here, to function application

```
fun nextNode: Node → Node {
  { n, m: Node | no next[n]
      implies m = first else m = next[n] } }
pred between [n1, nb, n2: Node] { // 'lt' is '<'
  lt[n1, n2] implies (lt[n1, nb] and lt[nb, n2])
    else (lt[n1, nb] or lt[nb, n2]) }
```

In Sect. II, we defined the key properties of Chord networks, called `Valid` and `Ideal`. The former is the conjunction of five properties: (1) there is *at least* one ring; (2) there is *at most* one ring; (3) any appendage node can reach a ring member by following successor pointers; (4) non-loopiness: there cannot be a ring member between a ring member and its successor; (5) successor list validity: the first successor of any member is between the member itself and the member's second successor.

```
pred valid { atLeastOneRing and atMostOneRing and
  orderedRing and connectedAppendages and
  orderedSuccessors }
pred atLeastOneRing { some ring }
pred atMostOneRing { all m1, m2: ring | m1 in m2.^succ }
pred connectedAppendages {
  all m1: appendages | some m2: ring | m2 in m1.^succ }
pred orderedRing { // = non−loopy
  all disj m1, m2, mb: ring |
  // 'disj' = 'all different'
    m2 = m1.succ implies not between[m1, mb, m2] }
pred orderedSuccessors { // successor list validity
  all m: members | between[m, m.fst, m.snd] }
```

An *ideal* network is a *valid* one *s.t.* (1) every member is in the ring (*i.e.* there are no appendages); (2) the `fst` and `prdc` functions are mutual inverses (local consistency) (3) the successor list of any member of the network contains the first 2 nodes that follow it in the network.

```
pred ideal {
  valid and no appendages and fst = ~prdc
  // '~' means 'transpose'
  all m: members { m.snd + m.fst in members
                   m.snd = m.fst.fst } }
```

C. Chord Events

1) An Action Layer: To model Chord events, we rely on an experimental *action* layer recently added to Electrum [16] that makes specification of transition systems much leaner. Actions are introduced by the keyword `act` and may take arguments. Their body is a conjunction of constraints referring to the current instant or the one following it immediately. The set of possible traces is automatically defined; notice that it implements (as of writing this article) an interleaving model of time: at every instant, exactly one action happens. Finally, an action comes with a `modifies` clause that contains the names of variable signatures and fields that the action may modify: an implicit fact then states that all other ones remain invariant under this action (this is usually called the action *frame condition*).

2) Communication Model: Following [4], our operations are atomic actions that may read *and* modify variables on at most two nodes. Compared to an asynchronous model, this is a *shared-state* abstraction that, in particular, hides the fact that nodes communicate through queued messages. Notice finally that the PODC paper states that communication is assumed to be reliable.

3) Events: The `join` action modifies `fst`, `snd` and `prdc` fields, and `members` and `ring` variable signatures. Under a `join` action, the joining node `new` must not be a member already. In PODC, the informal description of this event states that `new` contacts any node of the network and then makes a query to find a node `m` such that `new` is between `m` and its first successor. In our model, we abstract this query, assuming there is an oracle to determine this `m`. This abstraction does not affect the correctness of the protocol. Indeed, seeking the best position for the incoming node is an implementation and performance detail. Then `new` gets its pointers `fst` and `snd` from `m` and takes the latter as its predecessor[4].

```
act join [new: Node]
modifies fst, snd, prdc, members, ring {
  new not in members
  some m: members {
    between[m, new, m.fst] and fst' = fst ++ new→m.fst
    snd' = snd ++ new→m.snd and prdc' = prdc ++ new→m}}
```

Failures (or leavings) may happen too. When a member fails, it should empty all its fields. Besides, we take as an hypothesis that a node failure should not happen if it would leave another *member* with absolutely no live successors, meaning we forbid too many failures from happening on the same node to the point where it would completely break the network (this models the PODC failure assumptions: the protocol is indeed not able to fix networks split in several components that are mutually unreachable). Here, as there are only two successors per node, this requirement is easily modelled by stating that any node which points at the failing node using the `succ` relation keeps at least one of its two successors live when the failure happens.

Stabilization consists in fixing the first successor of a node. As in [4], stabilization is split here into two actions, depending on whether the concerned node has pending operations to do. This is shown in its `todo` field. We remark that, contrary to P. Zave, we may store *several* pending operations for a given node (otherwise, the field could be overwritten, which leads to a benign bug that we found during our analyses).

When there is not any pending operation for this node `m` (`stabilizeFromFst` action), it may contact its first successor. If the latter is dead (not a member), then `m` should update its `fst` field with its `snd` successor. The latter must also, in that case, be updated: to maintain the atomicity of the action, `m` should not contact any other node to get a new value for its `snd`. The solution here is just to take the immediate successor in the ring ordering. If it corresponds to no node, it will be fixed later by other events.

Otherwise, if the `fst` is a member, its value does not have to change but we update the `snd` as a small optimization. Besides,

[4]In the formalization, *e.g.* in the fourth line, `fst'` represents the value of `fst` in the *next* instant; `new→m.fst` is the pairing of `new` and `m.fst`; and `++` stands for the relational *override*. All in all, the line says that `fst'` is the current `fst` except in `new` where `fst'` will yield `m.fst`.

we check if fst's predecessor is not null and is better than m's first successor: if that is the case, the second form of stabilization (stabilizeFromFstPrdc) should be programmed; otherwise, m asks its first successor to program a future rectification with itself, meaning the first successor must ensure that m is its predecessor.

```
act stabilizeFromFst[m: Node]
modifies fst, snd, todo, members, ring {
  m in members
  no m.todo.Node // no pending operation
  m.fst not in members implies {
    todo' = todo and fst' = fst ++ m→m.snd
    snd' = snd ++ m→nextNode[m.snd] }
  else {
    fst' = fst and snd' = snd ++ m→m.fst.fst
    (some m.fst.prdc and between[m, m.fst.prdc, m.fst])
      implies todo' = todo + m→Stabilizing→m.fst.prdc
      else todo' = todo + m.fst→Rectifying→m } }
```

The second form of stabilization (stabilizeFromFstPrdc) may happen when a stabilization operation is pending: this is the case when a better candidate has been found for m.fst (during an stabilizeFromFst event). The first thing to do is to check whether the candidate would indeed, still, make a better fst. Besides, if the candidate is not even a member anymore, the operation must be cancelled. Otherwise, the fst field must be updated with the candidate (and the snd field is updated as well, with the candidate's fst field). Finally, this candidate is also told to rectify its predecessor later with m itself.

```
act stabilizeFromFstPrdc [m, newFst: Node]
modifies fst, snd, todo, members, ring {
  m in members and between[m, newFst, m.fst]
  m→Stabilizing→newFst in todo
  newFst not in members implies {
    todo' = todo − m→Stabilizing→newFst
    fst' = fst and snd' = snd }
  else {
    fst' = fst ++ m→newFst
    snd' = snd ++ m→ newFst.fst
    todo' = todo − (m→Stabilizing→newFst)
            + (newFst→Rectifying→m) } }
```

Finally, the rectify action aims at fixing a node m's predecessor: it may only happen if a rectification has been programmed. There are then three possibilities: (1) if m's predecessor is null or if the new candidate is better, then the predecessor should be updated to the candidate; (2) otherwise, if the current predecessor is not a member, the update is done too; (3) otherwise, m's predecessor is left as is.

D. Traces

As explained at the beginning of this section, the shape of traces is automatically set by the Electrum action layer. At any instant, exactly one event happens.

A Chord network must run indefinitely. Nevertheless when the network becomes ideal, if there are no more join and fail, the network will deadlock. To avoid this concern, we also add a skip action, which is a silent action that leaves everything unchanged and does nothing.

```
act skip {} // does nothing, modifies nothing
```

Notice that we also impose that, in every trace, there are always at least three live nodes: this is due to the fact that the network should always have a size strictly greater than the size of successor lists.

E. Initial State

Concerning the initial state, we specify that the ring is the ideal state (and no node has pending operations and non-member nodes do not have a predecessor).

```
fact init { no nonMembers.prdc and no todo and ideal }
```

Notice that it is stronger than the original claim made in the PODC paper (proved wrong in [6]). [4] exhibits an invariant stronger than validity. Although Electrum alleviates us from expressing such an invariant, we still need to ensure that the initial state satisfies it. Saying that the network starts in an ideal state avoids formulating that property explicitly: it is in our view not that strong an hypothesis as a Chord network may start with a very small size.

F. Correctness

1) Basic Properties: Using the Electrum Analyzer, we checked that the specification is consistent (*i.e.* it admits a model, in the logical sense) and that all branches of all actions are realizable.

2) Correctness Statement: The correctness property for the Chord protocol is a *liveness* property. The translation from the PODC paper is straightforward thanks to LTL: it states that if there are, eventually, never any join or fail events, then, eventually, the network will become ideal and remain so (recall the inital state is set in Sect. IV-E). This is expressed by an Electrum *assertion*:

```
assert correctness {
  (eventually always not (join or fail))
    implies eventually always ideal }
```

3) Fairness: Checking this assertion with the Electrum Analyzer yields a counterexample that manifests itself as the endless repetition of the skip action in some non-ideal state. This is to be expected as the correctness property is a *liveness* property: any action that may cause starvation to the ones meant to fix the network will be a problem. Classically, the solution is to add *fairness* constraints on the said actions. Here we use strong fairness constraints, saying for instance that if the guard of rectify is infinitely often satisfied, then the effect of rectify will be satisfied infinitely often:

```
pred rectifyEnabled[m, n: Node] {
  m in members and  m→Rectifying→n in todo }
fact fairness {
  all n, m : Node |
    (always eventually rectifyEnabled[n,m])
      implies (always eventually rectify[n,m]) }
```

We added such constraints for all stabilization and rectification actions. Doing so excludes the kind of starvation described above. Actually, it corresponds to a requirement in the PODC paper that says that nodes should perform these actions "periodically".

4) Corner Cases: During our analysis of correctness, we were able to find a few benign corner cases in P. Zave's model. They were all found in a matter of seconds, simply by checking the correctness property (the ease of finding them comes in our view from the fact that the use of the LTL layer of Electrum helps circumvent the risk of overlooking some verifications to be done). This led to make a few simple fixes w.r.t. her model (*e.g.* using a `todo` field to gather several pending operations instead of only one).

5) Liveness Bug: However, the correctness property is still wrong: checking the assertion in the Electrum Analyzer yields a trace with six time instants, the last one looping back in its predecessor (we recall that traces are infinite and represented as finite traces with a back loop from the last state to a former one). We present in Fig. 4 these last two steps only. In the first one, a `stabilizeFromFst[Node$3]` event is performed: `Node$3` contacts it immediate successor (which is in the ring) and learns from it that `Node$0` may be a better first successor. Then it programs a `stabilizeFromFstPrdc` action for itself and this new candidate.

In the following instant, the said action is performed but, as `Node$0` happens not be a member, the stabilization operation is cancelled. A rectification on `Node$1` would be needed here, for it to take `Node$3` as its predecessor, but a thorough analysis show that there is no way to trigger it by any of the stabilization actions.

6) Fixed Model: We fix this by adding another rectification action that is *not* triggered by other actions but done "periodically" by the nodes themselves (*i.e.* we also add a strong fairness constraint for this new action; note that such an operation was actually present in the original Chord papers). As nodes cannot guess who they should take as a new predecessor, they should just set their predecessor pointer to null if it points to a non member. The bet here is that, by other operations, the said node will eventually find a correct predecessor.

```
act rectifyNull[m: Node] modifies prdc, members, ring {
  m in members
  m.prdc not in members
    implies prdc' = prdc − m→m.prdc else prdc' = prdc }
```

Checking the correctness assertion, once this has been added, yields no counter-example anymore.

V. Evaluation of Results

This section presents the evaluation of various properties as well as that of the final correctness property with the Electrum Analyzer. The verification is performed on a GNU/Linux-based work station featuring an Intel Xeon E5-2699 providing 512 GB RAM (time-out was set to 5 h.).

Depending on the analyses to perform, we relied on the bounded and unbounded model-checking techniques (BMC and UMC) provided by the tool: the former relying on either a translation to BMC over Minisat (performed by an Electrum extension of Alloy's Kodkod pivot solver) or to the BMC mode of nuXmv (`check_ltlspec_bmc_inc` algorithm); and the latter through the ultimate compilation to the nuXmv model checker

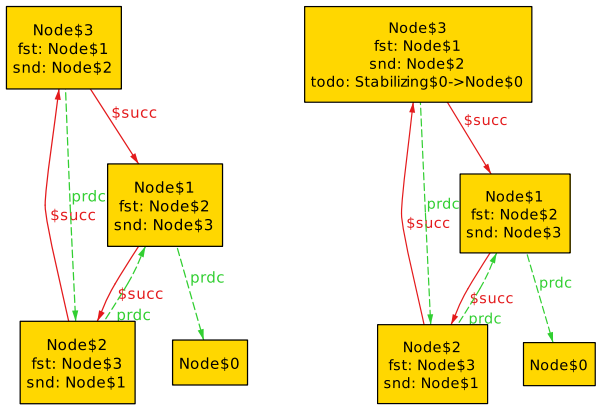

Fig. 4. Counterexample to `correctness` (loop part). Step 1 of the loop: a `stabilizeFromFst[Node$3]` is done; step 2 of the loop: a `stabilizeFromFstPrdc[Node$3, Node$0]`, then loop back to step 1.

TABLE I
Time (s.) for correctness analyses (buggy and fixed cases).

Prop.	Scope	M 10	M 15	XB 10	XB 15	XU
buggy	4	5	5	9	9	36
fixed	4	69	1 316	21	260	558
fixed	5	1 060	t/o	549	t/o	t/o
fixed	6	11 506	t/o	6 930	t/o	t/o
fixed	7	t/o	t/o	t/o	t/o	t/o

running the `check_ltlspec_klive` procedure [15]. As it is usually far more efficient, we always favored using the BMC technique when we were expecting to end up with an instance or a counter-example to a given property.

Although we do not report in detail on all properties due to lack of space, we first checked that our model is consistent (*i.e.* it admits an instance) and that all action branches are realizable. All these analyses ended positively in at most 10 s. using the BMC mode of the Electrum Analyzer.

We present in Table I the results for finding the liveness bug of Sect. IV-F5 and to check the correctness property for the fixed model (time in seconds; "t/o" means "time-out"; M is for BMC over Minisat, XB is for nuXmv in BMC mode, and XU is for nuXmv in UMC mode; for bounded modes, we considered bounds of 10 and 15 states).

As can be seen, correctness can be checked by the bounded analyzer for networks with 4-6 nodes and a time bound of 10 (4 nodes for a time bound of 15), while the unbounded Electrum analyzer yields a result for networks with 4 members only (taking into consideration that a basic ring already contains at least three nodes). This limitation in the size of the network with nuXmv is compensated by the fact that verification is exhaustive. For equal network sizes, the bounded version of Electrum Analyzer is faster than the unbounded one, as can be expected for valid properties. Finally, in bounded mode, nuXmv is faster than the Electrum implementation of BMC over Minisat, for the fixed model.

VI. Discussion

In this section, we would like to stress some important aspects of this work.

First, we modelled the Chord protocol in a very straightforward way thanks to the ease of use of Electrum. Compared with previous work, first-order relational logic, temporal logic and the action setting (with automatic handling of frame conditions and interleaving), combined with the push-button approach and visual feedback of the Electrum Analyzer, allowed us to quickly implement and test various approaches. Our model is inspired by important work of P. Zave, in particular its last incarnation [4], in that it essentially implements the same algorithm. But, we argue that our model is simpler, in particular because we do not have to deal with the details of state representation and because expressing complex temporal formulas over infinite traces is immediate.

Compared to P. Zave's analysis with SPIN [5], our modelling is also more straightforward as the author had to resort to various C programs to handle the *graph* notions present in Chord as well as visualization.

Second, due to this very reliance on LTL, we did not have to look for an inductive invariant to study the protocol. The search for the said invariant has been very arduous [4]–[8], but also illuminating: an invariant is not only a means of verification but also a way to understand a protocol better and to provide indications to developers. Notice also that Zave's invariant can actually be checked in Electrum exactly as in Alloy, with performances in the same order of magnitude. For these reasons, we think that Electrum may be used in early analysis and provide some help into finding the said invariant.

Third, we were able to find some corner cases in the Alloy model as well as the manual proof of [4] that were confirmed by P. Zave. In particular, the claimed invariant is indeed one but it is not strong enough to prove the protocol correctness. Fortunately, these issues were rather easy to fix.

Fourth, it is sometimes claimed that liveness "in the abstract" is not that important as what one really longs for is *bounded liveness*, which is actually a safety property. Our work confirms that straightforward temporal specification in LTL and pure liveness analysis are useful to find various issues (including a too weak invariant).

Finally, although limited to very small networks, our analysis is, up to our knowledge, the first "push-button" analysis of the actual correctness property of Chord, which is a liveness property.

We have insisted in this paper on P. Zave's work as this has been an important one but also because it served as a basis to lots of other works. However, the main recent work [17] relies on manual proofs in Coq and proves a safety property over Raft. [18] features an interesting mostly-automated approach but also focuses on an invariant proof. More recent work by the same authors [19] addresses liveness properties (for other protocols) but still with manual interaction.

In another line of work, [20] relied on π-calculus and for a bisimulation proof of the correctness of a very simple version of Chord (without failures). The proof was purely manual.

Besides, other distributed system protocols have been formally studied using "high-level" specification languages. For instance, Pastry was analyzed using TLA$^+$ [21] and other work used Event-B [22] to partly verify other protocols. However, these studies are limited to the verification of safety properties.

VII. Conclusion

This work presented the specification and verification of the Chord distributed protocol. We highlighted the usefulness of a lightweight modeling method that allows modeling and verifying dynamic systems with rich structural properties, as exemplified by Electrum (in particular with its action layer). Electrum allowed a simple and straightforward modeling of both structural and temporal properties of Chord, with a rather high abstraction level and without losing the key concepts of the protocol.

The analysis of the Chord model with the Electrum Analyzer is fully automated (on a bounded domain), which implies that the cost of entry is rather lower than many other formal methods. This analysis allowed us to find a few minor issues in the most important previous work (in which we took inspiration) and to show that the invariant there is not strong enough. We were able to fix all issues. Up to our knowledge, this is the first work analyzing the correctness of Chord, a liveness property, in a "push-button" way.

As of now, this analysis is admittedly limited in the size of networks, in particular for unbounded model checking. This was expected to us as one of the reasons to study Chord, for us, was to get a challenging test bed for our unbounded model-checking back-end. On the other hand, even with small networks and bounded model-checking, we were able to find various shortcomings in previous work, which confirms the interest on working even with small instances.

In the future, we will work both on improving Electrum and its analysis tools, and on the Chord protocol. For the former aspect, we will investigate smarter verification techniques, both on the bounded and unbounded sides (the latter is in particular, as of now, implemented in a naive way). For the latter aspect, we will investigate imperfect detection of failures.

Acknowledgment

The authors are grateful to Pamela Zave for her explanations and comments. We also thank Nuno Macedo for adding or fixing some Electrum Analyzer features needed for our study. Finally we thank the reviewers for their very useful remarks.

Work 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] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[2] D. Liben-Nowell, H. Balakrishnan, and D. Karger, "Analysis of the evolution of peer-to-peer systems," in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. ACM, 2002, pp. 233–242.

[3] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 1, pp. 17–32, 2003.

[4] P. Zave, "Reasoning about identifier spaces: How to make chord correct," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1144–1156, Dec 2017.

[5] ——, "A practical comparison of Alloy and SPIN," *Formal Aspects of Computing*, vol. 27, no. 2, p. 239, 2015.

[6] ——, "Using lightweight modeling to understand Chord," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 49–57, 2012.

[7] ——, "Why the Chord ring-maintenance protocol is not correct," AT&T Research, Tech. Rep, Tech. Rep., 2011.

[8] ——, "Lightweight Modeling of Network Protocols in Alloy," 2009.

[9] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations," in *Foundations of Software Engineering*, 2016.

[10] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[11] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[12] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.

[13] A. Cunha, "Bounded model checking of temporal formulas with Alloy," in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 303–308.

[14] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: A new symbolic model verifier," in *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, 1999, pp. 495–499.

[15] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 334–342.

[16] J. Brunel, D. Chemouil, A. Cunha, T. Hujsa, N. Macedo, and J. Tawa, "Proposition of an action layer for electrum," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2018, pp. 397–402.

[17] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, "Verdi: a framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 2015, pp. 357–368.

[18] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "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*, 2016, pp. 614–630.

[19] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, "Reducing liveness to safety in first-order logic," *PACMPL*, vol. 2, no. POPL, pp. 26:1–26:33, 2018.

[20] R. Bakhshi and D. Gurov, "Verification of peer-to-peer algorithms: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 181, pp. 35–47, 2007.

[21] S. Merz, T. Lu, and C. Weidenbach, "Towards Verification of the Pastry Protocol using TLA^+," in *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, vol. 6722, 2011.

[22] J. Risson, K. Robinson, and T. Moors, "Fault tolerant active rings for structured peer-to-peer overlays," in *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*. IEEE, 2005, pp. 18–25.