Towards a Verified Model of the Algorand Consensus Protocol in Coq

Musab A. Alturki[1], Jing Chen[2], Victor Luchangco[2], Brandon Moore[1],
Karl Palmskog[3], Lucas Peña[4], and Grigore Roşu[4]

[1] Runtime Verification, Inc., Urbana, IL, USA
{musab.alturki,brandon.moore}@runtimeverification.com
[2] Algorand, Inc., Boston, MA, USA
{jing,victor}@algorand.com
[3] The University of Texas at Austin, TX, USA
palmskog@acm.org
[4] University of Illinois at Urbana Champaign, IL, USA
{lpena7,grosu}@illinois.edu

Abstract. The Algorand blockchain is a secure and decentralized public ledger based on pure proof of stake rather than proof of work. At its core it is a novel consensus protocol with exactly one block certified in each round: that is, the protocol guarantees that the blockchain does not fork. In this paper, we report on our effort to model and formally verify the Algorand consensus protocol in the Coq proof assistant. Similar to previous consensus protocol verification efforts, we model the protocol as a state transition system and reason over reachable global states. However, in contrast to previous work, our model explicitly incorporates timing issues (e.g., timeouts and network delays) and adversarial actions, reflecting a more realistic environment faced by a public blockchain. Thus far, we have proved *asynchronous safety* of the protocol: two different blocks cannot be certified in the same round, even when the adversary has complete control of message delivery in the network. We believe that our model is sufficiently general and other relevant properties of the protocol such as liveness can be proved for the same model.

Keywords: Algorand · Byzantine consensus · blockchain · Coq.

1 Introduction

The Algorand blockchain is a scalable and permissionless public ledger for secure and decentralized digital currencies and transactions. To determine the next block, it uses a novel consensus protocol [1,3] based on pure proof of stake. In contrast to Bitcoin [6] and other blockchains based on proof of work, where safety is achieved by making it computationally expensive to add blocks, Algorand's consensus protocol is highly efficient and does not require solving cryptographic puzzles. Instead, it uses *cryptographic self-selection*, which allows each user to individually determine whether it is selected into the committees responsible for generating the next block. The self-selection is done randomly and independently

by every participant, with probability proportional to its stake. Private communication channels are not needed, and the committees propagate their messages in public. They reach Byzantine consensus on the next block and certify it, so that all users learn what the next block is without any ambiguity. That is, rather than waiting for a long time so as to be sure that a block will not disappear from the longest chain like in Bitcoin, the Algorand blockchain does not fork, a certified block is immediately final and transactions contained in it can be relied upon right away. The Algorand blockchain guarantees fast generation of blocks as long as the underlying propagation network is not partitioned (i.e., as long as messages are delivered in a timely fashion). The Algorand consensus protocol, its core technology, and mathematical proofs of its safety and liveness properties are described in [3,1,2].

The focus of this work is to formally model and verify the Algorand consensus protocol using the Coq proof assistant. Automated formal verification of desired properties adds another level of assurance about its correctness, and developing a precise model to capture the protocol's runtime environment and the assumptions it depends on is interesting from a formal-methods perspective as well. For example, [11] proves state machine safety and linearizability for the Raft consensus protocol in a non-Byzantine setting, and [7] focuses on safety properties of blockchains and, using a largest-chain-based fork-choice rule and a clique network topology, proves eventual consistency for an abstract parameterized protocol. Similar to existing efforts, in this work we define a transition system relation on global protocol states and reason inductively over *traces* of states reachable via the relation from some initial state. As in previous efforts, we abstract away details on cryptographic primitives, modeling them as functions with the desired properties.

However, our goal and various aspects of the Algorand protocol presented new challenges. First, our goal is to verify the protocol's asynchronous safety under Byzantine faults. Thus, we explicitly allow arbitrary adversarial actions, such as corruption of users and replay of messages. Also, rather than relying on a particular network topology, we explicitly model global time progression and message delivery deadlines in the underlying propagation network. In particular, the Algorand protocol assumes that messages are delivered within given deadlines when the network is not partitioned, and that messages may be arbitrarily delayed and their delivery is fully controlled by the adversary when the network is partitioned. We have captured these aspects in our model. Moreover, as mentioned above, the Algorand protocol uses cryptographic self-selection to randomly select committees responsible for generating blocks. As mechanizing probabilistic analysis is still an open field in formal verification, instead of trying to mechanize randomized committee selection, we identify properties of the committees that are used to verify the correctness of the protocol without reference to the protocol itself. We then express these properties as axioms in our formal model. Pen-and-paper proofs that these properties hold (with overwhelming probability) can be found in [3,1].

It is worth pointing out that our approach is based on reasoning about *global* states and allows an adversary to arbitrarily coordinate actions among corrupted users. This is different from [8], which formally verifies the PBFT protocol under arbitrary local actions. Finally, [10] uses distributed separation logic for consensus protocol verification in Coq with non-Byzantine failures. Using this approach to verify protocols under Byzantine faults is an interesting avenue of future work.

Thus far, we have proved in Coq the *asynchronous safety* property of the protocol: namely, two different blocks can never be both certified in the same round, even when the adversary has complete control of message delivery in the network. We believe that our model is sufficiently general and other relevant properties of the protocol such as liveness can be proved for the same model.

2   The Algorand Consensus Protocol

In this section, we give a brief overview of the Algorand consensus protocol with details salient to our formal model. More details can be found in [3,5,1].

All users participating in the protocol have unique identifiers (public keys). The protocol proceeds in *rounds* and each user learns a *certified* block for each round. Rounds are asynchronous: each user individually starts a new round whenever it learns a certified block for its current round.

Each round consists of one or more *periods*, which are different attempts to generate a certified block. Each period consists of several *steps*, in which users propose blocks and then vote to certify a proposal. Specifically, each user waits a fixed amount of time (determined by network parameters) to receive proposals, and then votes to support the proposal with the best *credential* as described below; these votes are called *soft-votes*. If it receives a quorum of soft-votes, it then votes to certify the block; these votes are called *cert-votes*. A user considers a block certified if it receives a quorum of cert-votes. If a user doesn't receive a quorum of cert-votes within a certain amount of time, it votes to begin a new period; these votes are called *next-votes*. A next-vote may be for a proposal, if the user received a quorum of soft-votes for it, or it may be *open*. A user begins a new period when it receives a quorum of next-votes from the same step for the same proposal or all being open; and repeats the next-vote logic otherwise.[5]

Committees. For scalability, not all users send their messages in every step. Instead, a committee is randomly selected for each step via a technique called *cryptographic self-selection*: each user independently determines whether it is in the committee using a verifiable random function (VRF). Only users in the committee send messages for that step, along with a *credential* generated by the VRF to prove they are selected. Credentials are totally ordered, and the ones accompanying the proposals are used to determine which proposal to support.

---

[5] The actual logic for next-votes is more complex, but roughly speaking the next-votes are classified as either for proposals or open.

Network. Users communicate by propagating messages over the network. Message delivery is asynchronous and may be out-of-order, but with upper bounds on delivery times. However, messages may not be delivered within these bounds if the network is *partitioned*.

Adversary. The adversary can corrupt any user and control and coordinate corrupted users' actions: for example, to resend old messages, send any message for future steps of the adversary's choice, and decide when and to whom the messages are sent by them. The adversary also controls when messages are delivered between honest users within the bounds described above, and fully controls message delivery when the network is partitioned. The adversary cannot, however, control more than 1/3 of the total stake participating in the consensus protocol.

3   Model

Our model of the protocol in the Coq proof assistant is in the form of a transition system, encoded as an inductive binary relation on global states. The transition relation is parameterized on finite types of user identifiers (`UserId`) and values (`Value`); the latter abstractly represents blocks and block hashes.

User and Global State. We represent both the user state and global state as Coq records. For brevity, we omit a few components of the user state in this paper and only show some key ones, such as the Boolean indicating whether a user is corrupt, the local time, round, period, step, and blocks and cert-votes that have been observed. The global state has the global time, user states and messages via finite maps [4], and a Boolean indicating whether the network is partitioned.

```
Record UState := mkUState {           Record GState := mkGState {
 corrupt: bool; timer: R;              network_partition: bool;
 round: ℕ; period: ℕ; step: ℕ;         now: R;
 blocks: ℕ → seq Value;                users: {fmap UserId → UState};
 certvotes: ℕ → ℕ → seq Vote;          msgs: {fmap UserId → {mset R * Msg}};
 (* ... omitted ... *)                 msg_history: {mset Msg};
}.                                    }.
```

State Transition System. The transition relation on global states g and g', written g ⤳ g', is defined in the usual way via inductive rules. For example, the rule for adversary message replay is as follows:

```
step_replay_msg : ∀ (pre:GState) uid (ustate_key : uid ∈ pre.(users)) msg,
  ¬ pre.(users).[ustate_key].(corrupt) → msg ∈ pre.(msg_history) →
  pre ⤳ replay_msg_result pre uid msg
```

Here, `replay_msg_result` is a function that builds a global state where `msg` is broadcasted. We call a sequence of global states a *trace* if it is nonempty and g ⤳ g' holds whenever g and g' are adjacent in the sequence.

Assumptions. To enable expressing relevant properties about our transition relation, we add assumptions about committees and quorums. This includes a function `committee` that determines self-selected committees, which we use to express properties of overlapping user quorums, as in the following statement, which says that for any two sets (quorums) of users of size at least `tau`, that are both subsets of the committee for the given round-period-step triple, there is an honest user for the step who belongs to both quorums:

```
Definition quorum_honest_overlap_statement (tau:ℕ) :=
 ∀ (trace:seq GState) (r p s:ℕ) (q1 q2:{fset UserId}),
  q1 ⊆ committee r p s → #|q1| ≥ tau →
  q2 ⊆ committee r p s → #|q2| ≥ tau →
  ∃ (honest_voter : UserId), honest_voter ∈ q1 ∧ honest_voter ∈ q2 ∧
   honest_during_step (r,p,s) honest_voter trace.
```

Similarly, we capture that a block was certified in a period as follows:

```
Definition certified_in_period (trace:seq GState) (tau r p:ℕ) (v:Value) :=
 ∃ (certvote_quorum:{fset UserId}),
  certvote_quorum ⊆ committee r p 3 ∧ #|certvote_quorum| ≥ tau ∧
  ∀ (voter:UserId), voter ∈ certvote_quorum →
   certvoted_in_path trace voter r p v.
```

This property is true for a trace if there exists a large-enough quorum of users selected for cert-voting who actually sent their votes along that trace for the given period (via `certvoted_in_path`, which we omit here). This is without loss of generality since a corrupted user who did not send its cert-vote can be simulated by a corrupted user who sent its vote but the message is received by nobody.

4 Asynchronous Safety

The analysis of the protocol in the computational model permits forking, albeit with negligible probability [1,3]. In contrast, we specify and prove formally in the *symbolic* model with idealized cryptographic primitives that at most one block is certified in a round, even in the face of adversary control over message delivery and corruption of users. We call this property *asynchronous safety*:

```
Theorem asynchronous_safety : ∀ (g0:GState) (trace:seq GState) (r:ℕ),
 state_before_round r g0 → is_trace g0 trace →
 ∀ (p1:ℕ) (v1:Value), certified_in_period trace r p1 v1 →
 ∀ (p2:ℕ) (v2:Value), certified_in_period trace r p2 v2 →
 v1 = v2.
```

Here, the first precondition `state_before_round r g0` states that no user has taken any actions in round `r` in the initial global state `g0`, and the second precondition `is_trace g0 trace` states that `trace` follows ⤳ and starts in `g0`.

Note that it is possible to end up with block certifications from multiple periods of a round. Specifically, during a network partition, which allows the adversary to delay messages, this can happen if cert-vote messages are delayed enough for some users to advance past the period where the first certification was produced. However, these multiple certifications will all be for the same block.

Proof Outline. The proof of asynchronous safety proceeds by case-splitting on whether the certifications are from the same period or different periods. For the first and the easier case, `p1 = p2`, we use quorum hypotheses to establish that there is an honest user that contributed a cert-vote to both certifications. Then, we conclude by applying the lemma `no_2_certvotes_in_p`, which establishes that an honest user cert-votes at most once in a period (proved by exhaustive analysis of possible transitions by an honest node):

```
Lemma no_2_certvotes_in_p : ∀ (g0:GState) (trace:seq GState) uid (r p:ℕ),
 is_trace g0 trace →
∀ idx1 v1, certvoted_in_path_at idx1 trace uid r p v1 →
  user_honest_at idx1 trace uid →
∀ idx2 v2, certvoted_in_path_at idx2 trace uid r p v2 →
  user_honest_at idx2 trace uid → idx1 = idx2 ∧ v1 = v2.
```

The second case ($p1 \neq p2$) is proved with the help of proving an invariant. This invariant first holds in the period that produces the first certification — say, `p1` for `v1`— and keeps holding for all later periods of the same round. The invariant is that no step of the period produces a quorum of open next-votes, and any quorum of value next-votes must be for `v1`.

5   Conclusion

We developed a model in Coq of the Algorand consensus protocol and outlined the specification and formal proof of its asynchronous safety. The model and the proof open up many possibilities for further formal verification of the protocol, most directly of *liveness* properties. In total, our Coq development [9] contains around 2000 specification lines and 4000 lines of proof scripts.

References

1. Algorand blockchain features (2019), `https://github.com/algorandfoundation/specs/blob/master/overview/Algorand_v1_spec-2.pdf`
2. Chen, J., Gorbunov, S., Micali, S., Vlachos, G.: Algorand Agreement: Super fast and partition resilient Byzantine agreement. Cryptology ePrint Archive, Report 2018/377 (2018), `https://eprint.iacr.org/2018/377`
3. Chen, J., Micali, S.: Algorand: A secure and efficient distributed ledger. Theoretical Computer Science **777**, 155–183 (2019)
4. Cohen, C.: finmap (2019), `https://github.com/math-comp/finmap`
5. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: SOSP. pp. 51–68 (2017)
6. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
7. Pîrlea, G., Sergey, I.: Mechanising blockchain consensus. In: CPP. pp. 78–90 (2018)
8. Rahli, V., Vukotic, I., Völp, M., Esteves-Verissimo, P.: Velisarios: Byzantine fault-tolerant protocols powered by Coq. In: ESOP. pp. 619–650 (2018)
9. Runtime Verification, Inc.: Algorand verification (2019), `https://github.com/runtimeverification/algorand-verification`
10. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. PACMPL **2**(POPL), 28:1–28:30 (2018)
11. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.: Planning for change in a formal verification of the Raft consensus protocol. In: CPP. pp. 154–165 (2016)