Verification of Algorand in the Coq Proof Assistant (Work-in-progress Report)

Brandon Moore
Runtime Verification, Inc.
brandon.moore@runtimeverification.com

Karl Palmskog
The University of Texas at Austin
palmskog@utexas.edu

Lucas Peña
Runtime Verification, Inc.
University of Illinois at Urbana-Champaign
lpena7@illinois.edu

Musab A. Alturki
Runtime Verification, Inc.
King Fahd University of Petroleum and Minerals
musab@kfupm.edu.sa

Grigore Roşu
Runtime Verification, Inc.
University of Illinois at Urbana-Champaign
grosu@illinois.edu

Abstract

Algorand is a distributed platform for building secure and reliable decentralized systems. In this report, we describe our effort to model and verify the Algorand consensus protocol in the Coq proof assistant. We give an overview of the protocol, outline our formalization of the protocol as a state transition system, and describe how its properties are formalized and proved. A major contribution of this work is the formalization of the assumptions under which the main safety theorem is proved. The Coq source files are available at:
https://github.com/runtimeverification/algorand-verification

1 Introduction

Algorand is a distributed platform for building secure and reliable decentralized systems, proposed by a world-renowned cryptographer and MIT professor, Silvio Micali At the heart of Algorand is the Algorand consensus protocol, a pure Proof-of-Stake protocol that promises to provide efficient, secure and scalable operation while remaining truly decentralized. The basic idea in pure PoS is to make the security of the system dependent only on how honest the majority of its assets are, without having to rely on any specific small subset of nodes or having to lock up assets and penalize users.

Consensus protocols in general are inherently complex. They involve interaction between many independent and potentially untrusted nodes in a large network where asynchronous and failure-prone communication is the norm rather than the exception. Furthermore, Algorand, in particular, describes not only the nodes' internal behavior, but also deals with message propagation and delivery, to be able to address two levels of attacks: (1) Attacks on the protocol, by corrupting participating nodes so that they no longer follow the protocol, e.g., to produce blocks with fake transactions or to cast votes for the wrong block, and (2) Attacks on the underlying network on which the system is deployed, for instance by intercepting, manipulating and delaying messages. When an attacker gains full control of message delivery in a network, the network is said to be partitioned. Most existing consensus protocols do not consider what happens when a network is partitioned or after the network recovers from a partition.

Consequently, ensuring correctness and resilience against malicious behavior while designing such a protocol is a challenging task, and its importance cannot be overemphasized. Since consensus systems are the backbone of large cryptocurrencies worth billions of US dollars, vulnerabilities can have potentially catastrophic consequences, as we have seen several times in the past. Having strong formal guarantees of correctness and developing a deep understanding of their assumptions can significantly reduce the possibility of such events happening.

To achieve the highest levels of assurance, we chose deductive verification, in which systems are modeled and specified inside expressive formal logical systems, and verified in a similar way to how mathematicians prove theorems - in principle by elaborating proofs of statements step-by-step. Among the large spectrum of formal techniques, deductive verification provides the strongest guarantees, and thus the highest degree of trustworthiness, but at the expense of being the most demanding.

We used Coq, which is a proof assistant based on type theory, for our deductive verification effort. Coq is developed for more than 30 years and very well supported. Coq has previously been used to develop a verified C compiler, formally prove mathematical results such as the four-color theorem, and verify distributed protocols (see, for example, our previous Casper formal verification effort for the Ethereum Foundation).

We developed a model of the Algorand consensus protocol in Coq, and proved a slew of its properties that we then used

Project Report, June 16, 2019, Runtime Verification, Inc.
2019.

to ultimately show the asynchronous safety property: no two honest nodes certify two different blocks, even when the adversary has complete control of message delivery in the network. This report describes our effort to model the protocol and specify and verify its properties. Moreover, we describe the formalization of the assumptions under which the safety theorem holds. Beyond the safety theorem, we intend for this formalization in Coq to lay the foundation for further future modeling and verification efforts of the Algorand consensus protocol.

2 Background

This section generally introduces blockchain and consensus protocols, gives an overview of Algorand, and provides pertinent Coq background.

2.1 Blockchains and Consensus Protocols

A blockchain is an ordered sequence of cryptographically linked blocks of records, in much the same way as links in a chain are firmly latched in sequence. A blockchain provides a persistent, tamper-proof and globally accessible ledger of transactions. It is built using well-established and publicly known cryptographic tools, including most importantly one-way hash functions [add pointer]. Anyone can verify the validity of its transactions, and no one can tamper with a transaction or claim a transaction that does not appear in the chain.

This makes blockchains particularly well suited for building self-governing and autonomous distributed systems. Indeed, a blockchain is a key component for allowing a collection of nodes in a communication network (who do not necessarily trust each other) to work collectively and make decentralized decisions. Nevertheless, the nodes will need more than just a blockchain to achieve proper decentralized operation. Specifically, they need a mechanism for identifying and agreeing on the next block of transactions to be appended to the chain, a mechanism known as a consensus protocol.

A consensus protocol typically proceeds in rounds, where the objective of a round is to try to produce the next block and record it in the chain. As such, a consensus protocol includes both: (1) a mechanism for decentralized selection of one or more block proposers or producers of a round in the protocol, and (2) a mechanism for achieving decentralized consensus on a single block to be appended to the chain.

In an idealistic (and rather unrealistic) setup, in which communication networks are perfectly reliable with zero-delay message delivery, and in which all participating nodes run error-free code and behave honestly, achieving consensus on a single block in each round is a trivial task. All nodes would instantaneously see all proposed blocks and their trusted opinions about them. In reality, however, the situation is actually very different. Distributed systems utilize global Internet (IP-based) communication networks, which are inherently unreliable and where significant message transmission delays are not uncommon. Furthermore, nodes may deviate from the protocol, either intentionally (when compromised by malicious users) or unintentionally (due to internal errors). These complications can hinder the consensus process, resulting potentially in the system being unusable or in losing large amounts of assets maintained by the system.

Therefore, consensus protocols need to maintain a consistent global view of the system while relying only on the local knowledge at the level of its individual (honest) nodes. Moreover, these protocols must ensure continuous and fair operation of the system in the presence of both benign node or network failures and maliciously behaving nodes. Consequently, consensus protocols are inherently complex, consisting of multiple asynchronous steps and utilizing a wealth of cryptographic and randomization schemes, and possibly some economic incentive structures, whose goal is to reward compliant behaviors and penalize deviations from the protocol.

A class of consensus protocols, pioneered by Bitcoin and referred to as Proof-of-Work (PoW), achieves consensus primarily through a process called mining, in which all participating nodes in the network compete on solving complex cryptographic puzzles to produce and record blocks. Although it provided workable solutions to the decentralized consensus challenge, PoW is now widely known to suffer from efficiency, scalability and security problems. The mining process is inherently computationally expensive, slow and wasteful, resulting in wasting significant amounts of energy [add example] while costing high transaction fees. Furthermore, PoW tends in practice to result in centralized systems, in which the bulk of the mining power (and hence the power of deciding the fate of the blockchain) falls in the hands of a small subset of users: those who can afford to invest in very powerful computing resources, or who can simply join forces in mining pools. Centralization defeats the purpose of the protocol and poses a major security concern: those who control the majority of the mining power need to be trusted, and even if they are, they make the system a much easier target for attacks.

An alternative mechanism that promises to alleviate these problems is Proof-of-Stake (PoS), in which the burden of producing blocks and selecting the next block to augment the blockchain is placed on a suitably selected subset of nodes (typically much smaller than the entire set of participating nodes), called a committee. In some variations of PoS, called delegated PoS, the committee is trusted and is usually fixed. In other variations, generally referred to as bonded PoS, participation in a committee requires staking some of the blockchain's underlying cryptographic assets (such as cryptocurrencies or tokens), which is the process of locking up these assets for an extended period of time so that they

cannot be expended or moved. The stake of a committee member (relative to the total stake in the system) determines the member's voting power when deciding the next block for a given round in the protocol. Furthermore, a node's stake can also be used as collateral to be (fully or partially) reclaimed by the system if a node is found to misbehave.

However, many important questions arise when designing a PoS protocol. How is the committee selected? One would want to have a selection process that is fair and representative, and hard to manipulate by a malicious user. Moreover, a committee that is fixed for a prolonged period of time can be easily attacked and presents a potential single point-of-failure for the system. But how often should the committee be changed? What is a suitable size of the committee in relation to the total population of nodes in the network? How should the voting process be designed? These and other design choices can significantly affect the efficiency, scalability and security of a PoS blockchain system.

2.2 Overview of Algorand

We highlight below some of Algorand's unique features. More detailed descriptions of these features and others can be found in [].

Algorand almost never forks. Forking happens when consensus on a single block for a round is not reached and multiple different blocks are added for that round. This is a notorious problem in PoW, but is also possible in PoS systems. Having different subsets of nodes decide on appending different blocks to the chain means that transactions appearing in these conflicting blocks are not finalized since only one of these blocks will eventually belong to the canonical chain, the chain that is deemed most accepted by the nodes (there are different methods for deciding the canonical chain, e.g. the longest chain in Bitcoin). Algorand avoids this problem by design: at most one block can receive the majority of votes in a round.

Algorand is very efficient. In Bitcoin and other PoW protocols, the rate at which blocks are produced (and hence the rate at which transactions are processed in the blockchain) is determined by the complexity of the cryptographic puzzle to be solved by the miners: the more complex the problem is, the longer it takes to produce a block. Although simplifying these puzzles would mean faster block production, the effective transaction processing rate would likely be adversely impacted as the forking rate will also increase. More miners will now compete to have their blocks in the chain and many more transactions will now belong to chains that end up being abandoned (they will have to be re-processed and attempted again). Being a PoS system, Algorand does not have this problem, and is in fact designed to produce a block every second. This high block production rate when combined with the fact that the chain forks only with negligible probability means that transactions appear very quickly in the chain, and once they do, they can immediately be considered final. This has the potential of allowing extremely high levels of scalability.

Algorand is very secure. Algorand considers security against two levels of attacks: Attacks on the protocol, which involve corrupting participating nodes so that they no longer follow the protocol, e.g., by producing blocks with fake transactions or by casting votes for the wrong block or casting multiple conflicting votes. Attacks on the underlying network on which the system is deployed, for instance by intercepting, manipulating and delaying messages. When an attacker gains full control of message delivery in a network, the network is said to be partitioned. Most existing consensus protocols do not consider what happens when a network is partitioned or after the network recovers from a partition. An attacker's goal is generally to manipulate the chain to reverse transactions or rewrite history (e.g. double-spend attacks), or to attempt to break consensus, leaving the system in an inconsistent state, or even cause the system to halt, where the chain can no longer grow.

The key to Algorand's security in its pure PoS protocol is maintaining decentralization at every step of the protocol. This is generally achieved through several techniques relating to how a committee, which is a group of nodes selected to make a decision on behalf of all nodes in the network, is managed. We highlight below key aspects of committee managment:

- Membership in a committee is decided through a unique process called cryptographic self-selection, which is run individually by each node. Essentially, a node locally runs a cryptographically fair and irrefutable lottery whose outcome (which can be verified by everyone else) decides membership in the committee.
- Committees are not fixed and change very frequently throughout execution of the protocol. In fact, every step of execution may have its own committee. Furthermore, committees can be of different sizes depending on the type of task the committee is entrusted to perform. For instance, the committee responsible for producing a block is typically much smaller than the committee that votes on blocks.
- Cryptographic self-selection is performed secretively by each node in isolation of all other nodes. Only the node knows whether it's part of the current-step committee, until it announces its membership with its vote or block to the network. An attacker has no way of knowing beforehand whether a node belongs to a committee. Once the node announces its membership, it will already be too late for an attacker to corrupt the node and send a different message. The node's original messages are already out and being propagated

through the network (note also these messages are digitally signed, so they cannot be tampered with without the receiver knowing).
- Another distinguishing feature of Algorand is the use of ephemeral keys, which are temporary, single-use encryption-decryption keys, for signing vote and block messages. Once a node uses an ephemeral key to sign a message, the node immediately destroys it, so that if the node gets compromised later on by an attacker, the attacker won't be able to claim that the node sent a different message for a previous step in the protocol. This effectively prevents an attacker from going back in history and forging different messages from what the nodes originally sent.

By performing cryptographic self-selection at every step of the protocol and using ephemeral keys for signing block and vote messages, Algorand decouples the execution of a protocol step from the identities of nodes participating in that step, a property referred to as player replaceability [add pointer]. Indeed, the execution of a step does not depend on the state of a node, and a node is never obliged to participate in a minimum number of steps in the protocol.

The Algorand consensus protocol proceeds in rounds and any node can participate in the protocol (i.e. Algorand is permissionless). A new block is generated at each round. A round is divided into periods, and a period is further divided into steps. A potential block is proposed in each period of a round, and the round ultimately ends when the block is finalized.

[add diagram]

The four kinds of steps in each period are the proposing step, filtering step, certifying step, and finally the finishing step(s). Present in each of these steps is Algorand's cryptographic self-selection process highlighted above. That is, a verifiable random function is used to privately and securely select committee members at each of these steps. The random, verifiable, self-selecting nature of this process is key to the correctness of the Algorand protocol.

Proposing step- Committee members in this first step (chosen by cryptographic self-selection) are known as potential leaders. These members propose a block and propagate it to other members of the protocol.

Filtering step- Here, committee members identify their leader, also using cryptographic self-selection. Committee members evaluate their current state, and potentially soft-vote for the proposed block.

Certifying step- In the certifying step, committee members simply evaluate if there are enough soft-votes for the potential block. If so, each committee member submits a cert-vote for that block. If there are enough certvotes for a block, the block is ultimately approved or certified, and users move to the next round.

Finishing steps- The last stage of a period, the finishing steps allow users to move to a new period without certifying a block. Here, committee members evaluate the proposed block, as well as votes received from other members, including votes in the last period. This allows users to potentially propagate a next-vote message. Sufficient next-votes result in the period increasing without a block being certified.

A node may proceed to the next period of a round when it observes a quorum of next-votes for a block (or for a special bottom value indicating that no potential block has enough votes yet). Once a node observes that a block gets a certificate, which is a large-enough quorum of cert-votes, the node certifies the block and moves on to the next round.

For a detailed and more precise description of these steps, the reader is referred to Algorand's technical reports[ADD LINKS].

2.3 Coq and Mathematical Components

One emerging approach to the challenge of designing and implementing correct and robust consensus protocols is to apply formal methods during design and development, which is the approach we advocate at RV. More specifically, we first formally specify both a protocol's design and the properties that the design is required to meet. Then, we mathematically verify that the design meets the requirements, or, if it does not, determine how the design may violate the requirements. In the latter case, we can use our findings to revise the design. When we successfully verify a design, we also precisely pin down the assumptions under which the design satisfies its requirements, which are extremely important to consider when deploying an implementation of a design and for systems built on top of an implementation.

The benefits of applying formal methods early in the design process of complex systems are well documented. Most directly, formal methods can uncover fundamental errors that would otherwise go undetected, and which are costly to correct in a later phase. Moreover, obtaining formal guarantees about a protocol's design significantly increases the confidence in not just the implementation of the protocol, but also in the systems built on top of it, and ultimately increases trust in whole platforms among developers and users, facilitating wider adoption and support.

Researchers have developed a large spectrum of techniques and tools for applying formal methods to model, specify, and verify general software systems and distributed protocols. The techniques vary across dimensions such as expressive power, automation, and tool support. Perhaps most importantly, they differ in the mathematical guarantees they provide and therefore in trustworthiness. Generally, to obtain stronger guarantees, more effort is required by formal verification engineers during both modeling and verification, and less automation is available (see the chart in Figure 1).

As the diagram indicates, the strongest guarantees, and thus the highest degree of trustworthiness, are obtained

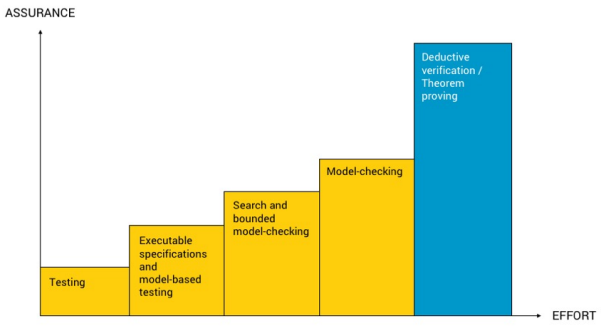

Figure 1. Different verification methods and how they generally compare with respect to the confidence-effort trade off.

through deductive verification. In deductive verification, systems are modeled and specified inside expressive formal logical systems, and verified in a similar way to how mathematicians prove theorems - in principle by elaborating proofs of statements step-by-step. Among tools for deductive verification, proof assistants offer the smallest trusted computing base necessary to trust verified statements, realized in modestly-sized trusted checkers that only assume a small set of axioms accepted by most mathematicians. Although they provide such strong guarantees, proof assistants can offer extensive automation and user support. Nevertheless, the process of figuring out what properties are relevant, how these properties are specified and building up their proofs normally requires a great deal of human ingenuity and experience. However, once everything is specified, proofs developed in proof assistants can be machine-checked, and persisted to serve as independently verifiable evidence that the properties hold for the given system.

Coq is a proof assistant based on type theory, developed for more than 30 years. Coq can be viewed as consisting of, on the one hand, a small and powerful purely functional programming language, and on the other hand, a system for specifying and proving properties. A Coq user writes functions and data and then interactively constructs the proof of a theorem by trying different proof tactics that transform the state of the in-progress proof. Coq only accepts the theorem after its checker has been run on the purported proof. Coq has previously been used to develop a verified C compiler, formally prove mathematical results such as the four-color theorem, and verify distributed protocols (see, for example, our previous Casper formal verification effort for the Ethereum Foundation).

[Mathematical Components]

3 Modeling and Verification Approach

[TODO: Karl] We developed a model of the Algorand consensus protocol in Coq, and proved a slew of its properties that we then used to ultimately show the asynchronous safety property. In total, our Coq development contains around 2000 lines of Gallina code for the specification of the model and its properties, and around 4000 lines of proof scripts that build the formal proofs. The source files also contain almost 1000 lines of comments that motivate and explain the specifications and lemmas.

The modeling effort involved primarily:

- Defining all the needed data types and structures, such as message types and records, using the inductive constructions facilitated by the logic underlying Coq. Definitions were designed to utilize various existing infrastructure components, provided as Coq libraries, including most prominently the Mathematical Components (MathComp) library. These definitions specify the structure of the protocol's state.
- Defining both: (1) utility functions, such as computing the number of soft-votes seen by a user so far or identifying the proposal with the least credential, and (2) logical propositions asserting, for example, the preconditions for a cert-voting action or whether some event (like cert-voting) had taken place along an execution path. Functions and propositions constitute the basic building blocks using which other more complex (inductive) definitions and statements are formed.
- Stating and proving basic structural properties about these definitions to ensure their correctness with respect to their intended behavior, for example, showing that a defined relation is a complete order or that a function that updates user timers preserves the set of users in the system. These properties are crucial for building up other proofs of more elaborate statements. We note that definitions imported from external libraries, such as MathComp, come equipped with many useful properties already proved, which are also heavily used in our model.
- Formalizing the argument for safety and proving the safety theorem, which in turn required designing a proof plan and identifying the intermediate results that will need to be shown as proof obligations. This process is iterative, since many intermediate results were deep enough that they required breaking them down into smaller, more manageable pieces. The whole process was guided by the informal proof arguments documented in the protocol description by Algorand[add pointers]. We highlight some of the details of this proof later in the report.

3.1 The State Transition Relation

At the heart of the model is the global state transition relation (GTransition defined here[add link] and denoted g > g', where g and g' are global states of the protocol). This relation defines inductively how the protocol's global state

transitions into another state in one step. For example, the global state may transition into another by advancing time, having one of its users make an internal transition or by having the adversary partition the network, and so on. Each transition is guarded by certain preconditions that need to be satisfied for the transition to be enabled. Therefore, the statement that g > g' tells us that the preconditions for one of these global transition cases were enabled and that transition was taken to transform g into g'.

Most statements asserting that some kind of event has taken place are specified as propositions on traces, possibly satisfying certain conditions. A trace is a non-empty sequence of global states that is built up using the global transition relation, i.e. if state g_i is the ith state in the sequence and g_{i+1} is the $(i+1)$th state, then the ordered pair g_i and g_{i+1} belongs to the global transition relation.

[add diagram]

By specifying path properties as propositions on traces, we are able to define generically what the property is without having to assume a concrete initial state (or a set of initial states). Any conditions required for the property to hold can be specified as constraints on the states of the trace being considered.

For example, the property that a block was certified in a period is captured by the proposition:

```
Definition certified_in_period trace r p v :=
  ∃ (certvote_quorum:{fset UserId}),
    certvote_quorum `<=` committee r p 3
  ∧ #|` certvote_quorum | >= tau_c
  ∧ ∀ (voter:UserId), voter ∈ certvote_quorum →
      certvoted_in_path trace voter r p v.
```

It states that the proposition holds for a trace if there exists a large-enough quorum of users selected for cert-voting who actually published cert-votes along that trace for the given period (the proposition `certvoted_in_path`).

3.2 Assumptions

Most formalizations of programs and protocols are parameterized in specific ways, to be valid across a wide range of configuration choices and scenarios that may occur in practice. Our Algorand model is parameterized on a finite set of users (e.g., network nodes), and on a finite set of values used in message payloads (representing blocks and block hashes).

```
Parameter UserId : finType.
Parameter Value : finType.
```

Values are assumed to be checkable for validity. We also consider an arbitrary, but totally ordered, datatype of credentials owned by users, and assume there is some way, given a credential, to say whether its owner is a committee member. Finally, we parameterize on delays and Algorand-specific configuration parameters such as number of votes required to move between periods. For example, this parameter represents the message delivery delay:

```
Parameter lambda : R.
```

Besides these and other parameters (16 in total), we also express assumptions, most related to our parameters, in the form of axioms. We use four kinds of axioms:

1. Standard axioms from classical logic. We use these axioms (three in total) to facilitate simpler use of and reasoning about real numbers.
2. Axioms about credentials (two in total), such as about their uniqueness across different users (the symbol <> means "not equal", and the arrow "->" denotes logical implication):

   ```
   Axiom credentials_different :
     ∀ (u u' : UserId) (r r' : nat) (p p' : nat) (s s' : nat),
     u ≠ u' → credential u r p s ≠ credential u' r' p' s'.
   ```

   This axiom states that any two distinct user identifiers u and u' will have distinct credentials no matter what the round, period and step values are.
3. Axioms on limits of message delays (two in total), such as the axiom specifying how the small message delivery delay lambda, block message delivery delay `big_lambda` and the recovery time period L are all related:

   ```
   Axiom delays_order : (3 * lambda <= big_lambda < L)%R.
   Axiom delays_positive : (lambda > 0)%R.
   ```

4. Axioms on the composition of groups (quorums) of users satisfying certain conditions. For example, the assumption that "any two large-enough cert-voting committees for a given round-period-step triple must share at least one honest user", which captures the honest supermajority assumption in Algorand for the cert-voting case, is specified as:

   ```
   Axiom quorums_c_honest_overlap : quorum_honest_overlap_statement tau
   ```

   where `quorum_honest_overlap_statement` is a definition parameterized by the committee size threshold value tau:

   ```
   Definition quorum_honest_overlap_statement (tau : nat) : Prop :=
     ∀ (trace : seq GState) (r p s : nat) (quorum1 quorum2 : {fset User
       quorum1 `<=` committee r p s →
       // quorum1 is a subset of the committee of r-p-s
       #|` quorum1 | >= tau →
       // quorum1 is at least as large as the threshold tau
       quorum2 `<=` committee r p s →
       // quorum2 is a subset of the same committee
       #|` quorum2 | >= tau →
       // quorum2 is at least as large as the threshold tau
       ∃ honest_voter,
         honest_voter ∈ quorum1
         ∧ honest_voter ∈ quorum2
         ∧ honest_during_step (r,p,s) honest_voter trace.
   ```

`quorum_honest_overlap_statement` defines a proposition that holds if for any two quorums of users, each of size at least tau, and who are all committee members for the given r-p-s triple, there is an honest user for the step r-p-s who belongs to both quorums. In total, there are six axioms of this kind.

Our axioms either capture statements accepted by most mathematicians (1 above) or assumptions made by Algorand designers on the runtime environment (2-4 above).

4 The Safety Theorem

[TODO: Brandon] The safety theorem says that only one block may be certified in a round, which means the blockchain will not fork. The key assumptions of the proof are some properties of cryptographic self-selection. We have not formalized the cryptographic reasoning, or the notion of negligible probability, and instead simply assume that it's impossible to have a quorum of committee members without enough honest members.

The statement of the theorem says that any two certificates from the same round must be for the same block:

```
Theorem safety: ∀ g0 trace (r : nat),
  state_before_round r g0 →
  is_trace g0 trace →
  ∀ p1 v1, certified_in_period trace r p1 v1 →
  ∀ p2 v2, certified_in_period trace r p2 v2 →
  v1 = v2.
```

Note the preconditions `state_before_round r g0`, which states that the trace goes back long enough in history when no user has yet entered round r, and `is_trace g0 trace`, which states that the trace is a valid trace beginning at g0, obtained by the transitive closure of the global transition relation.

When there is a "network partition" allowing the adversary to delay messages, it's possible to end up with certificates from multiple periods if cert-vote messages are delayed enough for some nodes to advance to the next period, but these certificates will still all be for the same block.

The proof of the main safety theorem above[LINK safety] first considers two cases, for whether the two certificates are from the same period or different periods.

The case when the certificates are in the same period is handled in the lemma `one_certificate_per_period`. This proof uses the quorum hypothesis to conclude that there is an honest node that contributed a cert-vote to both quorums, and the lemma `no_two_certvotes_in_p` shows that an honest node cert-votes at most once in a period, by analyzing all the steps of the protocol.

The case of different periods is proved using an invariant, which holds for the period that produces the first certificate and for all later periods in the same round. The invariant is defined in `period_nextvoted_exclusively_for`, and says that no step of these periods produce a quorum of open next-votes, and any quorum of next-votes is for the same value that the first certificate was for.

Lemma `certificate_is_start_of_next_period` shows that this is true of next votes from the period that produced the first certificate, Lemma `excl_enter_excl_next` shows that if this is true for one period, it is true for the next. Lemma `prev_period_nextvotes_limits_cert_vote` shows that if the invariant holds for one period then any certificate produced in the next period can only be for the same value.

The lemma `certificate_is_start_of_next_period` is the only place we need to use a more complicated quorum hypothesis. Intuitively, if committee members are selected approximately randomly from the full population of users, so if some property (unrelated to committee membership) is true for all honest members of one quorum, then it's almost certainly true of most honest users overall, and even for a later step there cannot be another quorum where none of the honest members have that property.

The specific property we consider is that the user had seen a quorum of soft-votes for value v by the end of the cert-voting step of the period where the first certificate was produced. This is a precondition for making an honest cert-vote, so this property is true for all honest users whose vote is part of the certificate. Then by the interquorum assumption we have that most honest users will have seen a quorum of soft-votes for v, and any quorum of next-voters from this period has at least one honest user that saw a quorum of soft-votes for v. By the preconditions for next-voting this honest user could only have next-voted for v, so the invariant is established.

To prove the latter two lemmas `excl_enter_excl_next` and `prev_period_nextvotes_limits_cert_vote` we consider that any quorum of next-votes or cert-votes contains an honest voter, and looking at the preconditions for an honest node to make a cert-vote or next-vote. In both cases the vote can only be for v because we know from the invariant that any quorum of soft-votes in the new period are votes for v by `prev_period_nextvotes_limits_honest_soft_vote`, and that our honest node hasn't received a quorum of open next-votes from the previous period by lemma `no_bottom_quorums_during_from_nextvoted` `prev_period_nextvotes_limits_honest_soft_vote` is proven in turn by looking at the rules for an honest node to advance to a new period to establish than the node got "starting value" v, and with that starting value it cannot soft-vote anything but v unless it has also seen a quorum of open next-votes.

5 Conclusion

The model we developed as part of this effort is generic, in that it captures the dynamics of the Algorand consensus protocol in a way that is orthogonal to the properties that we verify about it. This means that the model can be readily used to verify other properties of the system beyond asynchronous safety, including most importantly liveness. In fact,

we anticipate that many of the smaller results shown about the protocol and used in the proof of safety will also constitute essential ingredients of the liveness proving effort. Nevertheless, proving liveness will probably require showing additional results, especially those related to timely message delivery and networking partitioning, which were not necessarily needed for the safety argument. But the model already has these components (time, message delays and network partitioning) and going into investigating liveness (and perhaps other properties) should be a seamless continuation of the effort involved in this project.

Acknowledgments

We thank Jing Chen, Nickolai Zeldovich and Victor Luchango from Algorand for their help throughout the project. This work was funded by Algorand.