AleoBFT Formal Specification

Aleo Network Foundation

January 22, 2025

Abstract

We provide a preliminary formal specification of AleoBFT, the consensus protocol of the Aleo blockchain.

Disclaimer. This specification is work in progress. It is a living document that currently covers only some aspects of AleoBFT. Most notably, it does not cover dynamic committees and stake: committees are static, as in most other BFT literature. [CM] contains a more complete specification, along with correctness proofs, all verified in the ACL2 theorem prover. We plan to extend this document to incorporate material from that formalization, written in the generic mathematical notation used in this document.

1 Introduction

AleoBFT is the consensus protocol of the *Aleo blockchain* [APa]. AleoBFT is based on *Narwhal* [DKKSS22] and *Bullshark* [SGSKK22a, SGSKK22b], with extensions for *dynamic committees* with *staking*. AleoBFT is implemented in *snarkOS* [APb] (primarily) and *snarkVM* [APc] (for some functionality).

This document provides a formal specification of AleoBFT, along with formal proofs of correctness. This specification and proofs are closely based on the ones developed using the ACL2 theorem prover [CM]. The ones in this document are written in a generic mathematical notation, which is more widely accessible than the ACL2 language and libraries.

The rest of this section provides an informal overview of AleoBFT. Section 2 contains the formal specification of AleoBFT, as a labeled state transition system (this notion is defined at the beginning of that section). Section 3 contains theorems, with proof sketches, of correctness properties of AleoBFT, mainly formulated as invariants of the labeled state transition system; since fully formal proofs have been carried out in ACL2, here it suffices to provide proof sketches, especially when they are useful to understand the protocol better. Section 4 provides some concluding remarks. The mathematical notation used in this document is overviewed in Section A.

1.1 AleoBFT

AleoBFT is run by *validators* that communicate via messages over a network that is normally the Internet. Each validator has a unique *address* associated with a private key, which provides cryptographic signature and verification capabilities.

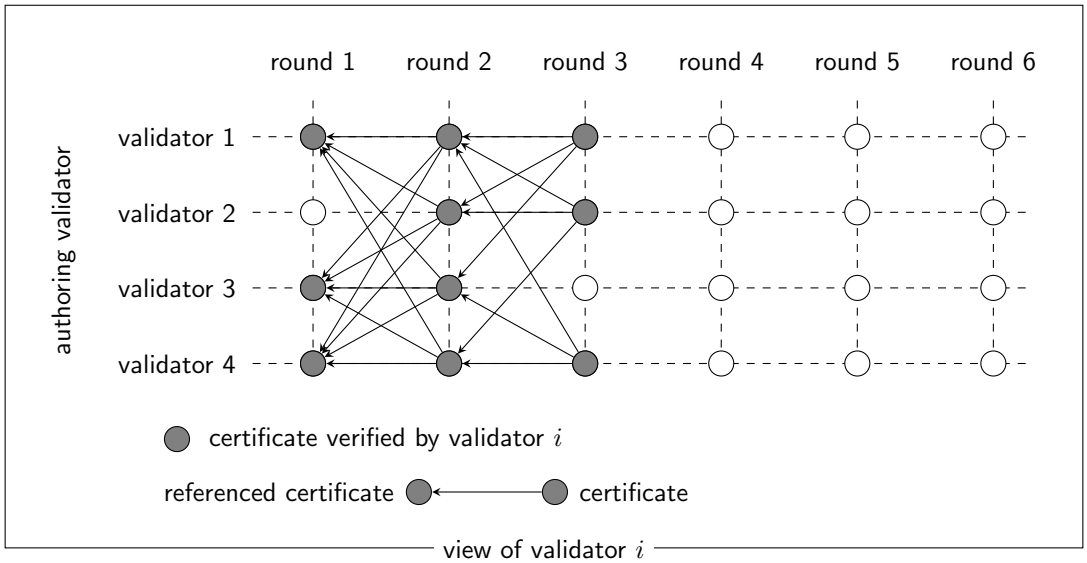
In the Narwhal part of AleoBFT, each validator collects *transactions*[1], from provers, clients, etc. When certain conditions hold, the validator *authors* (i.e. generates and signs) a *proposal* containing transactions, along with other data mentioned below, and broadcasts it to the other validators. Upon receiving a proposal, a validator independently validates it, including verifying the author's signature; if validation succeeds, the validator *endorses* the proposal by signing it and sending the signature back to the author. When a validator receives a certain number of endorsing signatures for a proposal it authored, the validator generates

[1] Transactions are one kind of *transmissions*, the other kinds being *solutions* and *ratifications*. It is transmissions, not just transactions, that are included in the blocks of the Aleo blockchain. However, in this formal specification, for simplicity we only talk of transactions, which we treat fairly abstractly anyhow, since they are a more standard term in the literature. Everything in this formal specification works if one mentally replaces 'transaction' with 'transmission'.

and broadcasts to the other validators a *certificate* containing the proposal and all the signatures. Upon receiving a certificate, a validator verifies the signatures and stores the certificate.

Validators store certificates in a DAG (directed acyclic graph), which can be visualized as a rectangular grid whose rows correspond to validators and whole columns correspond to *rounds* numbered 1, 2, etc. The certificates are the nodes of the DAG: each certificate has one author (the row) and one round number (the column); a validator authors at most one certificate per round. The edges of the DAG go from each certificate to a certain number of certificates in the previous round; as a special case, certificates in round 1 have no outgoing edges, since there is no round 0.

Figure 1: Example DAG



The information about each edge is part of the source certificate of the edge: each certificate (except in round 1) has references to certificates in the previous round, so if the certificate is in a validator's DAG, there are edges in that same DAG from the certificate to every certificate it references. These references are part of the proposal contained in the certificate: when a validator generates a proposal, it must already have a sufficient number of certificates in the DAG at the previous round, and the validator includes references to those certificates in the proposal. When the validator generates a certificate for the proposal (after receiving enough signatures, as explained above), it stores the certificate in its DAG and creates the described edges in its DAG. When a validator receives a certificate authored by another validator, it stores the certificate in its DAG after all the certificates referenced in the previous round are already in its DAG (requesting any missing certificates from other validators). So, in all cases, if a validator stores a certificate in its DAG (whether authored by the same or another validator), the edges from that certificate to certificates in the previous round are created at the same time the certificate is stored.

Validators start in round 1, where they can readily author proposals since no edges to previous certificates are required. When a validator has stored a certain number of certificates in round 1, it can move to round 2 and author a proposal at that round, and so on. To adapt to network delays that affect certificate delivery, validators can also advance their current round under other conditions.

Although validators may never quite have the same DAGs of certificates, the protocol's requirements on signatures and edges ensure that there is enough consistency across the DAGs of different validators that they can agree on a common partial order of (a subset of) their certificates, according to the edges of the DAG, and that if a certificate with a certain author and round is present, it is the same certificate for all validators. This is the critical *non-equivocation* property guaranteed by the Narwhal part of AleoBFT, as formally proved in this formal specification.

In the Bullshark part of AleoBFT, each validator extends their copy of the blockchain based on their own DAG, when certain conditions are satisfied. Each certificate has a *causal history*, which is the set of all

certificates in the DAG reachable from the certificate via the edges. Under certain conditions, a certificate in the DAG is *committed* to the blockchain: all the certificates in that certificate's causal history, except the ones already incorporated into the blockchain, are ordered in a canonical way (known to all validators), and their transactions are put into a new block that extends the blockchain.

A new block may be potentially created for each even round (2, 4, etc.), but some even rounds may be skipped. Each even round has one of the validators as *leader*: the leader for each even round is chosen via a deterministic computation known to all validators, which each validator can carry out locally. If a DAG has a certificate at an even round authored by the leader, that certificate is an *anchor* that may be committed to the blockchain. The anchor is committed if it has a sufficient number of incoming edges from certificates at the immediately subsequent odd round: these edges are considered *votes* for the anchor; the anchor is committed if it has enough votes.

The requirements on votes for anchors, together with the general requirements on the number of edges to previous certificates for every proposal discussed earlier, guarantee that all validators commit the same anchors in the same order. This is the critical *non-forking* property guaranteed by the Bullshark part of AleoBFT, as formally proved in this formal specification.

2 Definition

We model AleoBFT as a *labeled state transition system*, which in general is a tuple

$$Sys = \langle S, E, I, T \rangle$$

where:

1. S is the set of possible *states*.
2. E is the set of possible *events*.
3. $I \subseteq S$ is the set of possible *initial states*.
4. $T \subseteq S \times E \times S$ is the *transition relation* among old states, events, and new states.

Each triple $\langle s, e, s' \rangle \in T$ means that the event e causes a transition from the old state s to the new state s'.

Starting from an initial state $s_0 \in I$, a sequence of events $[e_1, e_2, \ldots]$ moves the system through a sequence of states $[s_1, s_2, \ldots]$, provided that $\langle s_{i-1}, e_i, s_i \rangle \in T$ for every i. The corresponding sequences of events and states may be infinite (if events are always possible) or finite (if a state is reached where events are no longer possible); an event e is possible in a state s when there exists a triple $\langle s, e, s' \rangle \in T$ for some s'. The system is nondeterministic in general: there may be multiple events possible in a state, and each such event may lead to different states; the relational nature of T (as opposed to being a function) accommodates this. The 'labeled' attribute of the state transition system refers to the fact that the transition relation is a ternary relation, involving not only old and new states but also events, as opposed to just a binary relation between old and new states: the transition from an old state to a new state is 'labeled' by the event.

In the rest of this section, we define a specific labeled transition system that models AleoBFT. Section 2.1 defines the possible states. Section 2.2 defines the possible events. Section 2.3 defines some operations on states and their components. Section 2.4 defines the possible initial states. Section 2.5 defines the possible state transitions. Section 2.6 puts everything together into the labeled state transition system.

2.1 States

The possible states of the system are built out of components, introduced here in a bottom-up fashion.

2.1.1 Addresses

We assume a set of possible *addresses*

$$Addr$$

which we leave abstract in this formal specification.

In the AleoBFT implementation, these are Aleo addresses `aleo1...` associated to private keys whose owners operate in the Aleo blockchain. In particular, each validator in AleoBFT has a private key and thus an address, which uniquely identifies the validator.

A correct validator is assumed not to share its private key, so a signature by a correct validator can only be produced by that validator. A faulty validator may share its private key, so a signature by a faulty validator may be produced by other parties with whom the private key was shared.

2.1.2 Round Numbers

We define the set of possible *round numbers* as

$$Round \triangleq \mathbb{N} \setminus \{0\}$$

i.e. positive integers (1, 2, 3, etc.).

AleoBFT is a DAG-based consensus protocol, where the DAG is organized in rounds denoted by increasing sequential integers starting from 1.

2.1.3 Transactions

We assume a set of possible *transactions*

$$Trans$$

which we leave abstract in this formal specification.

In the AleoBFT implementation there are transmissions, which include proper transactions as well as solutions and ratifications. The abstract notion of transactions in this formal specification captures all of those.

2.1.4 Blocks

We define the set of possible *blocks* as

$$\begin{aligned} Block \triangleq \{\mathsf{B}\langle \overline{trans}, round\rangle \mid\ & \overline{trans} \in Trans^* \\ & \wedge\ round \in Round\} \end{aligned}$$

i.e. a block consists of:

1. A finite sequence of transactions $\overline{trans}$.
2. A round number *round*.

In the AleoBFT implementation these are Aleo blockchain blocks, which have a much richer structure. But the definition above suffices for our formal specification, whose purpose is to show that transactions are properly put into blocks. As defined later, each block is produced in a round, and thus each block has an associated round number in a natural way; we explicate that in our definition above, since it is convenient for our formal specification.

2.1.5 Certificates

We define the set of possible *certificates* as

$$\begin{aligned} Cert \triangleq \{\mathsf{C}\langle auth, round, \overline{trans}, prevs, endors\rangle \mid\ & auth \in Addr \\ & \wedge\ round \in Round \\ & \wedge\ \overline{trans} \in Trans^* \\ & \wedge\ prevs \in \mathcal{P}_\omega(Addr) \\ & \wedge\ endors \in \mathcal{P}_\omega(Addr)\} \end{aligned}$$

i.e. a certificate consists of:

1. An address *auth* that denotes the author.
2. A round number *round*.
3. A finite sequence of transactions $\overline{trans}$.
4. A finite set *prevs* of addresses of authors of certificates in the previous round.
5. A finite set *endors* of addresses of validators who endorse the certificate.

Each certificate is produced by a validator (with address *auth*) at a certain round (with number *round*). This validator is called the *author* of the certificate. Correct validators produce at most one certificate per round; faulty validators are prevented by the protocol (as proved later) from producing more than one certificate per round. Thus, each certificate in the system is uniquely identified by its author *auth* and round number *round* (as proved later).

Note that we do not formally model proposals (proposed certificates) other than recording $\langle auth, round \rangle$ pairs in a validator's state (see Section 2.1.6) for the proposals the validator has endorsed.

A certificate contains (a sequence of) zero or more transactions $\overline{trans}$, which the author is proposing for inclusion in the blockchain.

Each certificate has references to a number of certificates in the previous round. Since the certificate includes its round number *round*, the previous round number $round - 1$ is known from that, so to identify a certificate in the previous round it suffices to use that certificate's author, since, as explained above, each certificate has a unique combination of author and round number. Thus, the *prevs* component of a certificate identifies the referenced certificates in the previous round.

For the special case $round = 1$, there is no previous round ($round - 1 = 0$ is not an actual round number): as formalized later, $prevs = \emptyset$ in this case, i.e. the certificate has no references to certificates in the previous round. For the more usual case $round > 1$, the cardinality of *prevs* is the quorum number (defined later). This chosen number of references to previous certificates ensures a number of desired properties, which are presented later.

The author of a certificate signs it with their private key. Our formal specification does not model signatures explicitly, but the presence of the *auth* component in the certificate models the presence of that signature. The author's signature attests that the certificate is valid: the transactions are valid, it is the only certificate created by the author for that round, the referenced previous certificates are in the validator's possession, and so on. If the author is correct, the attestation is assumed truthful. If the author is faulty, the attestation may not be truthful, so to prevent an invalid certificate from being generated, additional signatures are required.

This is why a certificate includes additional signatures by other validators, represented by the set of addresses *endors* of those validators. Those validators endorse a proposal by signing it: their signatures attest that the proposal is valid according to their independent validation. As proved later, endorsers are distinct from the author, i.e. $auth \notin endors$. The cardinality of the set of signers $\{auth\} \cup endors$ (i.e. the number of signatures) is the quorum number (defined later). As proved later, this number ensures that faulty authors cannot generate invalid certificates even if they get other faulty validators to sign their proposals. Similarly, an endorser other than the author may be correct or faulty, which affects the assumed truthfulness of the endorsement implied by its signature.

2.1.6 Validator States

We define the set of possible *validator states* as

$$\begin{aligned} Vstate \triangleq \{\mathsf{V}\langle round, dag, buf, ears, last, blocks, comm, timer\rangle \mid\ & round \in Round \\ & \wedge\ dag \in \mathcal{P}_\omega(Cert) \\ & \wedge\ buf \in \mathcal{P}_\omega(Cert) \\ & \wedge\ ears \in \mathcal{P}_\omega(Addr \times Round) \\ & \wedge\ last \in Round \cup \{0\} \\ & \wedge\ blocks \in Block^* \\ & \wedge\ comm \in \mathcal{P}_\omega(Cert) \\ & \wedge\ timer \in \{\mathsf{running}, \mathsf{expired}\}\} \end{aligned}$$

i.e. a validator state consists of:

1. A round number *round*.
2. A DAG *dag* that consists of a set of certificates.
3. A buffer *buf* that consists of a set of certificates.
4. A set *ears* of pairs of addresses and round numbers keeping track of endorsed proposals.
5. An indication *last* of the number of the last committed round, or 0 if no round has been committed.
6. A blockchain *blocks* that consists of a finite sequence of blocks.

7. A set *comm* of all the committed certificates.
8. An indication *timer* of whether the timer is running or has expired.

This models the internal state of a correct validator. As defined later, faulty validators have no explicit internal state in our formal specification. The following explanations apply to correct validators.

Each validator is in a round, starting from 1. The state component *round* indicates the round. It is incremented under certain conditions, as the validator advances through rounds.

Each validator has a DAG of certificates, which is modeled as the set of certificates *dag*. Under invariants proved later, a set of certificates indeed represents a DAG: the certificates are the vertices of the graph, and the edges are represented by the *prevs* components of the certificates, as explained in Section 2.1.5. The DAG can be visualized in a grid where rows correspond to validators (in some immaterial order) and columns correspond to round numbers (increasing from left to right): the row-column "coordinates" of each certificate correspond to its author and round number. The graph has a directed edge from each certificate with author *auth* and round number $round > 1$ to each certificate with author *prev* and round number $round - 1$ when *prev* is an element of the *prevs* state component of the first certificate. See figure 1 for an example.

The set *buf* of certificates is a buffer where each certificate received by a validator is initially put. The certificate is moved to the DAG *dag* only when the DAG has all the previous certificates linked by the certificate. This ensures that the DAG has all the outgoing edges of each certificate, so that it is closed under the causal history of each certificate; i.e. it has all the certificates reachable from each certificate via the edges of the DAG. Certificates may not arrive in order, so a certificate may stay in the buffer for a while. When the AleoBFT implementation receives a certificate, it actively requests the missing certificates from the certificate's causal history; our formal specification models those requests as separate events.

The *ears* state component is a set of pairs $\langle auth, round \rangle$ that identify proposals (from other validators), that this validator has endorsed (by signing) but has not yet received as a certificate. The exact use of *ears* is described later, when the transitions of the system are defined, but the purpose is to ensure non-equivocation of certificates, i.e. that there is at most one certificate with a given author and round number.

Each validator has its view *blocks* of the blockchain, as a sequence of blocks that is initially empty (we do not explicitly model the genesis block) and that grows from left to right.

Each validator keeps track of the last round in which it committed, to its blockchain, transactions from certificates in the DAG; that is, transactions from some of the certificates in the DAG are put into a new block that is added to the blockchain. As defined later, each commitment happens at a round, whose number is part of the block created by the commitment, as explained in Section 2.1.4; commitments happen at increasing rounds. The *last* state component is the number of the last round at which a block was committed. Initially the blockchain is empty, no commitment has happened yet, and $last = 0$, which is not a valid round number since round numbers are positive, as defined in Section 2.1.2.

Each validator keeps track of all the certificates that have been committed so far, in the *comm* state component. This facilitates the process of committing more certificates in this formal specification, which must exclude already committed certificates, as defined later.

Under certain situations, a validator starts a timer, which runs for a while until it eventually expires. We do not model real time in this formal specification, but we model the existence of the timer in two possible states, running and expired, and its transitions between these two states.

2.1.7 Messages

We define the set of possible *messages* as

$$Msg \triangleq \{\mathsf{M}\langle cert, dest\rangle \mid cert \in Cert \\ \wedge\ dest \in Addr\}$$

i.e. a message consists of:

1. A certificate *cert* that constitutes the payload of the message.
2. The address *dest* of the validator that is the destination of the message.

In this formal specification, this is the only kind of messages exchanged among validators. Since this model is currently focused on the Bullshark aspects of AleoBFT, the Narwhal aspects are modeled at an abstract level: instead of explicitly modeling the exchange of proposals and signatures that eventually lead

to certificates that are also exchanged, we model certificate creation as an atomic event, as the end result of the underlying exchange of proposals and signatures.

As defined later, each message is sent by the author of the certificate in the message. Thus, every message has explicit source and destination.

2.1.8 System States

We define the set of possible *system states* as

$$\begin{aligned} \mathit{State} \triangleq \{\mathsf{S}\langle \mathit{vstates}, \mathit{msgs}\rangle \mid\ & \mathit{vstates} : \mathit{Addr} \rightsquigarrow \mathit{Vstate} \cup \{\mathsf{faulty}\} \\ & \wedge\ \mathcal{D}(\mathit{vstates}) \neq \emptyset \\ & \wedge\ \mathit{msgs} \in \mathcal{P}_\omega(\mathit{Msg})\} \end{aligned}$$

i.e. a system state consists of:

1. A finite non-empty map *vstates* representing all the validators in the system and their states.
2. A set *msgs* of messages representing the state of the network.

AleoBFT is run by a non-empty collection of validators, whose addresses are the elements of the domain $\mathcal{D}(\mathit{vstates})$ of the map *vstates*. Each validator with address $\mathit{val} \in \mathcal{D}(\mathit{vstates})$ is: either *correct*, in which case $\mathit{vstates}(\mathit{val}) \in \mathit{Vstate}$, i.e. the validator has an associated internal state; or *faulty*, in which case $\mathit{vstates}(\mathit{val}) = \mathsf{faulty}$, i.e. the validator has an associated indication of faultiness. (Note that $\mathsf{faulty} \notin \mathit{Vstate}$.)

As defined later, validators never change their correct or faulty status: a validator is either always correct or always faulty. The 'always correct' case means that the validator always follows the protocol, while the 'always faulty' case means that the validator does not always follow the protocol, i.e. it is the negation of the first case. Any deviation from the protocol, no matter how small or temporary, renders a validator faulty. So 'always faulty' must be read not as never following the protocol, but failing to follow the protocol perfectly at any point, which is enough to mar the correctness status of the validator in a way that the formal specification considers permanent. This is normally the meaning of 'faulty' in the BFT literature, although that meaning may not be always explicated in the terms just explained.

The faulty designation for faulty validators does not model their internal states, which is appropriate for our formal model. Faulty validators may act arbitrarily, but what matters in the protocol is the impact that they may have on correct validators, specifically whether they can compromise the ability of correct validators to run the protocol and agree successfully. The only way for faulty validators to influence correct validators, in our model of the protocol, is to send messages to the network, which may be received by correct validators. Thus, we model (in the definition of state transitions) the possible messages that faulty validators may generate, but there is no need to have those messages depend on any specific internal state of the faulty validators.

The arbitrary behavior of faulty validators is of course constrained not to violate physical laws and such. In particular, faulty validators cannot break cryptography; as explained in Section 2.1.1, we assume that correct validators do not share their private keys. In addition, we assume that faulty validators cannot guess or extract correct validators' private keys and thus cannot forge their signatures. On the other hand, as notes in Section 2.1.1, faulty validators may well share their private keys.

2.2 Events

We define the set of possible *events* as

$$\begin{aligned} \mathit{Event} \triangleq\ & \{\mathsf{Ecreate}[\mathit{cert}] \mid \mathit{cert} \in \mathit{Cert}\} \\ \cup\ & \{\mathsf{Ereceive}[\mathit{msg}] \mid \mathit{msg} \in \mathit{Msg}\} \\ \cup\ & \{\mathsf{Estore}[\mathit{cert}, \mathit{val}] \mid \mathit{cert} \in \mathit{Cert} \wedge \mathit{val} \in \mathit{Addr}\} \\ \cup\ & \{\mathsf{Eadvance}[\mathit{val}] \mid \mathit{val} \in \mathit{Addr}\} \\ \cup\ & \{\mathsf{Ecommit}[\mathit{val}] \mid \mathit{val} \in \mathit{Addr}\} \\ \cup\ & \{\mathsf{Etimer}[\mathit{val}] \mid \mathit{val} \in \mathit{Addr}\} \end{aligned}$$

i.e. an event is one of the following:

1. The creation of a new certificate *cert* by its author.

2. The receipt of a message *msg* from the network to the destination validator.
3. The storage of a certificate *cert* into the DAG of a validator *val* from its buffer.
4. The advancement of the current round by a validator *val*.
5. The commitment of certificates by a validator *val*.
6. The timer expiration in a validator *val*.

Section 2.5 defines the exact circumstances in which each event is possible, and how the event changes the system state. The short descriptions above should give an initial intuitive idea.

2.3 Operations

2.3.1 Number of Validators

We define a function

$$\begin{aligned} &\mathit{NumAll} : \mathit{State} \to \mathbb{N} \setminus \{0\} \\ &\mathit{NumAll}(\mathsf{S}\langle \mathit{vstates}, \mathit{msgs} \rangle) \triangleq |\mathcal{D}(\mathit{vstates})| \end{aligned}$$

that yields the *total number of validators* in a system state, i.e. the cardinality of the domain of the validator map. As proved later, this domain is unchanged by state transitions.

This number is known to every validator. This is often referred to as n or N in the BFT literature.

2.3.2 Number of Correct Validators

We define a function

$$\begin{aligned} &\mathit{NumCorrect} : \mathit{State} \to \mathbb{N} \\ &\mathit{NumCorrect}(\mathsf{S}\langle \mathit{vstates}, \mathit{msgs} \rangle) \triangleq |\{\mathit{val} \in \mathcal{D}(\mathit{vstates}) \mid \mathit{vstates}(\mathit{val}) \neq \mathsf{faulty}\}| \end{aligned}$$

that yields the *number of correct validators* in a system state, i.e. the cardinality of the subset of the domain of the validator map associated to validator states and not to faulty. As proved later, this subset is unchanged by state transitions.

This number is known in the model, but not to validators.

2.3.3 Number of Faulty Validators

We define a function

$$\begin{aligned} &\mathit{NumFaulty} : \mathit{State} \to \mathbb{N} \\ &\mathit{NumFaulty}(\mathsf{S}\langle \mathit{vstates}, \mathit{msgs} \rangle) \triangleq |\{\mathit{val} \in \mathcal{D}(\mathit{vstates}) \mid \mathit{vstates}(\mathit{val}) = \mathsf{faulty}\}| \end{aligned}$$

that yields the *number of faulty validators* in a system state, i.e. the cardinality of the subset of the domain of the validator map associated to faulty and not to validator states. As proved later, this subset is unchanged by state transitions.

This number is known in the model, but not to validators. This is sometimes referred to as m or f in the BFT literature, but this symbol is also used to refer to the number we define in Section 2.3.4.

It is the case that

$$\mathit{NumAll}(\mathit{state}) = \mathit{NumCorrect}(\mathit{state}) + \mathit{NumFaulty}(\mathit{state})$$

2.3.4 Maximum Number of Faulty Validators

We define a function

$$\begin{aligned} &\mathit{MaxFaulty} : \mathit{State} \to \mathbb{N} \\ &\mathit{MaxFaulty}(\mathit{state}) \triangleq \max\{x \in \mathbb{N} \mid x < \mathit{NumAll}(\mathit{state})/3\} \end{aligned}$$

that yields the *maximum number of faulty validators* in a system state for the protocol to be Byzantine-fault-tolerant. This is the largest integer below one third of the total number; that is, more than two thirds of the validators must be correct. This is a classic condition in the BFT literature.

This number is sometimes referred to as m or f in the literature, which does not always emphasize the distinction between the maximum tolerated number *MaxFaulty* of faulty validators, which is calculable solely from *NumAll* and known to validators, and the actual number *NumFaulty* of faulty validators, which is unknown to validators (see Section 2.3.3).

It is the case that

$$MaxFaulty(state) = \lfloor (NumAll(state) - 1)/3 \rfloor$$

and that

$$MaxFaulty(state) = \lceil NumAll(state)/3 \rceil - 1$$

when $NumAll(state) > 0$, providing alternative definitions of *MaxFaulty* in terms of floor and ceiling of division.

Using the symbols n for $NumAll(state)$ and f for $MaxFaulty(state)$, as often done in the literature, since f is the maximum integer below $n/3$, there are three possible values for n as a function of f: $n = 3f + 1$, $n = 3f+2$, and $n = 3f+3$. Some BFT literature just uses the more restrictive assumption $n = 3f+1$, but that is not fully general; while it is normally not difficult to generalize results obtained under the more restrictive assumption to results for the more general assumption, caution must be exercised (see Section 2.3.6).

The definition of *State* in Section 2.1.8, requires the presence of at least one validator, so $NumAll(state) \neq 0$. The definition of *MaxFaulty* would be ill-formed otherwise. Indeed, the case of no validator is so degenerate as to not be of interest. If $NumAll(state) \in \{1, 2, 3\}$, then $MaxFaulty(state) = 0$, i.e. no failures are tolerated. The protocol could theoretically run with 1, 2, or 3 (all correct) validators, but in practice it is normally run with at least 4 validators, in order to tolerate at least one failure.

2.3.5 Fault Tolerance

We define a predicate

$$BFT : State \to \mathbb{B}$$
$$BFT(state) \triangleq [NumFaulty(state) \leq MaxFaulty(state)]$$

that says whether the system is *Byzantine-fault-tolerant* in a state: the actual number of faulty validators must not exceed the maximum number of faulty validators. Since *NumAll* is unchanged by state transitions, so is *MaxFaulty*, and so is the *BFT* predicate.

2.3.6 Quorum

We define a function

$$Quorum : State \to \mathbb{N} \setminus \{0\}$$
$$Quorum(state) \triangleq NumAll(state) - MaxFaulty(state)$$

that yields the *quorum number* used in the definition of the state transitions. This number is known to validators, since it is the difference between two known numbers: the total number of validators, and the maximum number of faulty validators (which is calculated from the total number as defined in Section 2.3.4). Like the numbers from which it is defined, the quorum number is unchanged by the state transitions.

Using the symbols n for $NumAll(state)$ and f for $MaxFaulty(state)$, as often done in the literature, the quorum number is $n - f$. As explained in Section 2.3.4, there are three possible values of n as a function of f: $n = 3f + 1$, $n = 3f + 2$, and $n = 3f + 3$. In the first case, $n - f = 2f + 1$; but in the other two cases, $n - f = 2f + 2$, or $n - f = 2f + 3$. Some BFT literature is not general or precise regarding the quorum number: it uses $2f+1$ instead of $n-f$, without explicating the assumption $n = 3f+1$ behind $2f+1$; it uses $2f + 1$ in the description of algorithms that are meant to be general and work for every n, when it should use $n-f$ in those algorithms instead. The general and correct quorum number is $n-f$, not $2f+1$, which is only correct under the (unnecessarily restrictive) condition that $n = 3f + 1$; the claimed theorems are false (i.e. they have counterexamples) if $2f + 1$ is used without the assumption that $n = 3f + 1$.

2.3.7 Validator State Updates

We define the following functions that update components of the validator state.

Function that *updates the round* of a validator state:

$$\begin{aligned}
&\mathit{UpdateRound} : \mathit{Vstate} \times \mathit{Round} \to \mathit{Vstate} \\
&\mathit{UpdateRound}(\mathit{vstate}, \mathit{round}') \triangleq \mathsf{V}\langle \mathit{round}', \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle \\
&\quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle
\end{aligned}$$

Function that *updates the DAG* of a validator state:

$$\begin{aligned}
&\mathit{UpdateDag} : \mathit{Vstate} \times \mathcal{P}_\omega(\mathit{Cert}) \to \mathit{Vstate} \\
&\mathit{UpdateDag}(\mathit{vstate}, \mathit{dag}') \triangleq \mathsf{V}\langle \mathit{round}, \mathit{dag}', \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle \\
&\quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle
\end{aligned}$$

Function that *updates the buffer* of a validator state:

$$\begin{aligned}
&\mathit{UpdateBuf} : \mathit{Vstate} \times \mathcal{P}_\omega(\mathit{Cert}) \to \mathit{Vstate} \\
&\mathit{UpdateBuf}(\mathit{vstate}, \mathit{buf}') \triangleq \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}', \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle \\
&\quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle
\end{aligned}$$

Function that *updates the endorsed pairs* of a validator state:

$$\begin{aligned}
&\mathit{UpdateEar} : \mathit{Vstate} \times \mathcal{P}_\omega(\mathit{Addr} \times \mathit{Round}) \to \mathit{Vstate} \\
&\mathit{UpdateEar}(\mathit{vstate}, \mathit{ears}') \triangleq \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}', \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle \\
&\quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle
\end{aligned}$$

Function that *updates the last committed round* of a validator state:

$$\begin{aligned}
&\mathit{UpdateLast} : \mathit{Vstate} \times (\mathit{Round} \cup \{0\}) \to \mathit{Vstate} \\
&\mathit{UpdateLast}(\mathit{vstate}, \mathit{last}') \triangleq \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}', \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle \\
&\quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle
\end{aligned}$$

Function that *updates the blockchain* of a validator state:

$$\begin{aligned}
&\mathit{UpdateBlocks} : \mathit{Vstate} \times \mathit{Block}^* \to \mathit{Vstate} \\
&\mathit{UpdateBlocks}(\mathit{vstate}, \mathit{blocks}') \triangleq \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}', \mathit{comm}, \mathit{timer}\rangle \\
&\quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle
\end{aligned}$$

Function that *updates the committed certificates* of a validator state:

$$\begin{aligned}
&\mathit{UpdateComm} : \mathit{Vstate} \times \mathcal{P}_\omega(\mathit{Cert}) \to \mathit{Vstate} \\
&\mathit{UpdateComm}(\mathit{vstate}, \mathit{comm}') \triangleq \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}', \mathit{timer}\rangle \\
&\quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle
\end{aligned}$$

Function that *updates the timer* of a validator state:

$$\begin{aligned}
&\mathit{UpdateTimer} : \mathit{Vstate} \times \{\mathsf{running}, \mathsf{expired}\} \to \mathit{Vstate} \\
&\mathit{UpdateTimer}(\mathit{vstate}, \mathit{timer}') \triangleq \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}'\rangle \\
&\quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \mathit{ears}, \mathit{last}, \mathit{blocks}, \mathit{comm}, \mathit{timer}\rangle
\end{aligned}$$

2.3.8 Leaders

We assume a function

$$\begin{aligned} &\mathit{Leader} : \mathcal{P}_\omega(\mathit{Addr}) \times \mathit{Round} \to \mathit{Addr} \\ &\text{WITH } \mathit{Leader}(\mathit{vals}, \mathit{round}) \in \mathit{vals} \end{aligned}$$

that picks, for each round, a *leader* address among a set of addresses.

When this function is used, *vals* is the set of all the validator addresses in the system, and the round number *round* is an even one (but we regard the function as defined on all rounds, for simplicity). The picked leader may be correct or faulty.

This function is known to every validator, so correct validators always agree on which validator is the leader at which round. But the details of the leader choice are immaterial in this specification.

2.3.9 Anchors

We define a function

$$\begin{aligned} &\mathit{Anchors} : \mathcal{P}_\omega(\mathit{Cert}) \times \mathcal{P}_\omega(\mathit{Addr}) \to \mathcal{P}_\omega(\mathit{Cert}) \\ &\mathit{Anchors}(\mathit{dag}, \mathit{vals}) \triangleq \{\mathsf{C}\langle \mathit{Leader}(\mathit{vals}, \mathit{round}), \mathit{round}, \ldots\rangle \in \mathit{dag} \mid \mathit{round} \bmod 2 = 0\} \end{aligned}$$

that yields the *anchors* of a DAG, i.e. the certificates at an even round authored by the leaders of those rounds.

Anchors are the certificates that, under suitable conditions, are committed to the blockchain along with their causal histories. A DAG does not necessarily have an anchor at each even round; the leader address is defined for every round, but a certificate authored by that leader in that round is not necessarily generated or received at any point in time. Even when an anchor is present in a DAG, it does not necessarily get committed: certain conditions must be satisfied, defined in Section 2.5.

2.3.10 Edges

We define a predicate

$$\begin{aligned} &\mathit{Edge} : \mathit{Cert} \times \mathit{Cert} \to \mathbb{B} \\ &\mathit{Edge}(\mathit{cert}, \mathit{cert}') \triangleq \begin{array}{l} \mathit{cert} = \mathsf{C}\langle \ldots, \mathit{round}, \ldots, \mathit{prevs}, \ldots\rangle \\ \land\ \mathit{round} > 1 \\ \land\ \mathit{cert}' = \mathsf{C}\langle \mathit{prev}, \mathit{round} - 1, \ldots\rangle \\ \land\ \mathit{prev} \in \mathit{prevs} \end{array} \end{aligned}$$

that says whether there is an *edge* from a certificate to another: the first certificate must be in a round after the first round, and the second certificate must be in the immediately preceding round, and the second certificate's author must be one of the addresses referenced by the first certificate. Note that this predicate is defined without explicit reference to a DAG, based only on the two certificates.

2.3.11 Paths

We define a predicate

$$\begin{aligned} &\mathit{Path} : \mathit{Cert} \times \mathit{Cert} \times \mathcal{P}_\omega(\mathit{Cert}) \to \mathbb{B} \\ &\mathit{Path}(\mathit{cert}, \mathit{cert}', \mathit{dag}) \triangleq \exists [\mathit{cert}_1, \ldots, \mathit{cert}_p] \in \mathit{Cert}^+ \;.\; \begin{array}{l} \mathit{cert}_1 = \mathit{cert} \\ \land\ \mathit{cert}_p = \mathit{cert}' \\ \land\ \forall i \in \{1, \ldots, p-1\} \;.\; \mathit{Edge}(\mathit{cert}_i, \mathit{cert}_{i+1}) \\ \land\ \forall i \in \{1, \ldots, p\} \;.\; \mathit{cert}_i \in \mathit{dag} \end{array} \end{aligned}$$

that says whether there is a *path* from a certificate to another in a DAG: there must exist a non-empty sequence of certificates in the DAG that starts with the first certificate and ends with the second certificate, where each pair of adjacent certificates in the sequence is linked by an edge. This definition includes the empty path from a certificate to itself, when $p = 1$ and $\mathit{cert}_1 = \mathit{cert} = \mathit{cert}'$.

2.3.12 Anchor Paths

We define a predicate

$$\begin{array}{l} \mathit{AnchorPath} : \mathit{Round} \times \mathit{Round} \times \mathcal{P}_\omega(\mathit{Cert}) \times \mathcal{P}_\omega(\mathit{Addr}) \to \mathbb{B} \\ \mathit{AnchorPath}(\mathit{round}, \mathit{round}', \mathit{dag}, \mathit{vals}) \triangleq \begin{array}{l} \mathit{cert} = \mathsf{C}\langle \ldots, \mathit{round}, \ldots \rangle \in \mathit{Anchors}(\mathit{dag}, \mathit{vals}) \\ \land\ \mathit{cert}' = \mathsf{C}\langle \ldots, \mathit{round}', \ldots \rangle \in \mathit{Anchors}(\mathit{dag}, \mathit{vals}) \\ \land\ \mathit{Path}(\mathit{cert}, \mathit{cert}', \mathit{dag}) \end{array} \end{array}$$

that says whether two rounds have anchors connected by a path: *cert* is an anchor at round *round* in the DAG, *cert′* is an anchor at round *round′* in the DAG, and there is a path from *cert* to *cert′* in the DAG. When this predicate is used, $\mathit{round}' < \mathit{round}$, i.e. we consider paths to earlier rounds.

2.3.13 Certificate Ordering

We assume a function

$$\begin{array}{l} \mathit{Order} : \mathcal{P}_\omega(\mathit{Cert}) \to \mathit{Cert}^* \\ \text{WITH } \left(\begin{array}{l} \mathit{Order}(\mathit{certs}) = [\mathit{cert}_1, \ldots, \mathit{cert}_m] \Longrightarrow \\ \mathit{certs} = \{\mathit{cert}_1, \ldots, \mathit{cert}_m\} \land (\forall i, i' \in \{1, \ldots, m\}\ .\ i \neq i' \Longrightarrow \mathit{cert}_i \neq \mathit{cert}_{i'}) \end{array} \right) \end{array}$$

that orders a set of certificates into a sequence without repetitions, according to some criterion that is immaterial to this specification.

As formalized later, this is used by (correct) validators to sequentialize certificates to commit to the blockchain. All validators use the same ordering function assumed here.

2.4 Initialization

We define the set of possible *initial system states* as

$$\begin{array}{l} \mathit{InitState} \triangleq \{\mathsf{S}\langle \mathit{vstates}, \emptyset \rangle \mid \forall \mathit{val} \in \mathcal{D}(\mathit{vstates})\ .\ \mathit{vstates}(\mathit{val}) = \mathsf{V}\langle 1, \emptyset, \emptyset, \emptyset, 0, \epsilon, \emptyset, \mathsf{expired} \rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lor\ \mathit{vstates}(\mathit{val}) = \mathsf{faulty}\} \end{array}$$

i.e. there are no messages in the network, and each correct validator has round number 1, no certificates in the DAG, no certificates in the buffer, no endorsed author-round pairs, no blocks, no committed rounds, no committed certificates, and the timer expired (i.e. not running). There are different possible initial states for every choice of addresses of correct and faulty validators: given those, *vstates* must assign to each such address either the initial validator state or the faulty indication.

2.5 Transitions

We define the transitions of the system by defining, for each kind of event:

1. A predicate that says under which conditions an event can happen in a state.
2. A function that says which new state results from that event, when it happens.

Given a state and an event on which the predicate holds, there is thus a unique new state to which the system moves, namely the one returned by the function. However, different events may be possible in the same state, making the system nondeterministic.

2.5.1 Certificate Creation

We define a predicate

$$\begin{array}{l} \mathit{HasNoAR} : \mathit{Vstate} \times \mathit{Addr} \times \mathit{Round} \to \mathbb{B} \\ \mathit{HasNoAR}(\mathit{vstate}, \mathit{auth}, \mathit{round}) \triangleq \begin{array}{l} \mathsf{C}\langle \mathit{auth}, \mathit{round}, \ldots \rangle \notin \mathit{dag} \\ \land\ \mathsf{C}\langle \mathit{auth}, \mathit{round}, \ldots \rangle \notin \mathit{buf} \\ \land\ \langle \mathit{auth}, \mathit{round} \rangle \notin \mathit{ears} \end{array} \\ \quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \ldots, \mathit{dag}, \mathit{buf}, \mathit{ears}, \ldots \rangle \end{array}$$

that says whether a validator does not have a given author and round, in the form of a certificate with that author and round in the DAG or buffer, or in the form of an element in the set of endorsed pairs. This predicate is used below to express the condition that (correct) validators sign (i.e. author or endorse) a certificate only if it is a new certificate from the given author in the given round, according to the information that validators have recorded in their state.

We define a predicate

$$\begin{array}{l} \mathit{HasPrevious} : \mathit{Vstate} \times \mathcal{P}_\omega(\mathit{Addr}) \times \mathit{Round} \to \mathbb{B} \\ \mathit{HasPrevious}(\mathit{vstate}, \mathit{prevs}, \mathit{round}) \triangleq \forall \mathit{prev} \in \mathit{prevs} \,.\, \mathsf{C}\langle \mathit{prev}, \mathit{round} - 1, \ldots \rangle \in \mathit{dag} \\ \quad \text{WHERE} \;\; \mathit{vstate} = \mathsf{V}\langle \ldots, \mathit{dag}, \ldots \rangle \end{array}$$

that says whether a validator has, in its DAG, all the certificates with given authors at the round immediately preceding a given round. This predicate is used below to express the condition that (correct) validators sign (i.e. author or endorse) a certificate only if they have all the certificates that the new certificate links to. This predicate is used only with $\mathit{prevs} = \emptyset$ if $\mathit{round} = 1$: in that case, the universal quantification is trivially true, without the need to consider certificate forms $\mathsf{C}\langle \ldots, 0, \ldots \rangle$ that are technically ill-formed.

We define a predicate

$$\begin{array}{l} \mathit{CreateCondAuth} : \mathit{Vstate} \times \mathit{Cert} \to \mathbb{B} \\ \mathit{CreateCondAuth}(\mathit{vstate}, \mathit{cert}) \;\triangleq\; \mathit{round}' = \mathit{round} \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge \;\; \mathsf{C}\langle \mathit{auth}, \mathit{round}, \ldots \rangle \notin \mathit{dag} \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge \;\; \mathit{HasPrevious}(\mathit{vstate}, \mathit{prevs}, \mathit{round}) \\ \quad \text{WHERE} \;\; \mathit{vstate} = \mathsf{V}\langle \mathit{round}', \mathit{dag}, \ldots \rangle \\ \qquad\quad \wedge \;\; \mathit{cert} = \mathsf{C}\langle \mathit{auth}, \mathit{round}, \ldots, \mathit{prevs}, \ldots \rangle \end{array}$$

that says whether a new certificate can be created from the standpoint of the author (if correct):

1. The round number of the certificate must be the author's current round number.
2. The DAG must not contain an old certificate with the same author and round number.
3. The DAG must have all the certificates that the new certificate links to.

Correct validators author at most one certificate per round, and do so only if their DAG has all the previous certificates. The predicate *HasPrevious* is defined earlier.

We define a predicate

$$\begin{array}{l} \mathit{CreateCondEndor} : \mathit{Vstate} \times \mathit{Cert} \to \mathbb{B} \\ \mathit{CreateCondEndor}(\mathit{vstate}, \mathit{cert}) \;\triangleq\; \mathit{HasNoAR}(\mathit{vstate}, \mathit{auth}, \mathit{round}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge \;\; \mathit{HasPrevious}(\mathit{vstate}, \mathit{prevs}, \mathit{round}) \\ \quad \text{WHERE} \;\; \mathit{cert} = \mathsf{C}\langle \mathit{auth}, \mathit{round}, \ldots, \mathit{prevs}, \ldots \rangle \end{array}$$

that says whether a new certificate can be created from the standpoint of an endorser (if correct):

1. The endorser must not already have a record of the author and round of the certificate.
2. The endorser's DAG must have all the certificates that the new certificate links to.

Correct endorsers sign at most one certificate per author per round, and do so only if their DAG has all the previous certificates. The predicates *HasNoAR* and *HasPrevious* are defined earlier.

We define a predicate

$$
\begin{array}{l}
\mathit{CreateCond} : \mathit{State} \times \mathit{Cert} \to \mathbb{B} \\
\mathit{CreateCond}(\mathit{state}, \mathit{cert}) \triangleq \mathit{auth} \in \mathcal{D}(\mathit{vstates}) \\
\qquad \land \ \mathit{endors} \subseteq \mathcal{D}(\mathit{vstates}) \\
\qquad \land \ \mathit{auth} \notin \mathit{endors} \\
\qquad \land \ |\mathit{endors}| = \mathit{Quorum}(\mathit{state}) - 1 \\
\qquad \land \ |\mathit{prevs}| = \begin{cases} 0 & \text{if } \ \mathit{round} = 1 \\ \mathit{Quorum}(\mathit{state}) & \text{otherwise} \end{cases} \\
\qquad \land \ \begin{pmatrix} \mathit{vstates}(\mathit{auth}) \neq \mathsf{faulty} \implies \\ \mathit{CreateCondAuth}(\mathit{vstates}(\mathit{auth}), \mathit{cert}) \end{pmatrix} \\
\qquad \land \ \forall \mathit{endor} \in \mathit{endors} \ . \ \begin{pmatrix} \mathit{vstates}(\mathit{endor}) \neq \mathsf{faulty} \implies \\ \mathit{CreateCondEndor}(\mathit{vstates}(\mathit{endor}), \mathit{cert}) \end{pmatrix} \\
\quad \text{WHERE} \ \ \mathit{state} = \mathsf{S}\langle \mathit{vstates}, \ldots \rangle \\
\qquad \land \ \mathit{cert} = \mathsf{C}\langle \mathit{auth}, \mathit{round}, \ldots, \mathit{prevs}, \mathit{endors} \rangle
\end{array}
$$

that says whether a certificate may be created in a state:

1. The certificate author must be a validator in the system.
2. The certificate endorsers must be validators in the system.
3. The author must be distinct from the endorsers.
4. The number of endorsers must be one less than the quorum.
5. The certificate must have links to a quorum of previous certificates, or to none if the round is 1.
6. If the author is correct, its state satisfies the conditions for certificate creation.
7. For each correct endorser, its state satisfies the conditions for certificate creation.

The conditions expressed by *CreateCondAuth* and *CreateCondEndor* apply only to correct authors and endorsers. Faulty ones do not have an internal state, and can author and endorse certificates at will, so long as the certificates can get enough signatures. In general, author and endorsers together form a quorum (this critically depends on the condition that the author is distinct from the endorsers). The condition $\mathit{prevs} \subseteq \mathcal{D}(\mathit{vstates})$, i.e. that the authors of the previous certificates are validators in the system, is not included in the above definition because it follows from *HasPrevious* and invariants proved later.

We define a function

$$
\begin{array}{l}
\mathit{CreateNextAuth} : \mathit{Vstate} \times \mathit{Cert} \to \mathit{Vstate} \\
\mathit{CreateNextAuth}(\mathit{vstate}, \mathit{cert}) \triangleq \mathit{UpdateDag}(\mathit{vstate}, \mathit{dag} \cup \{\mathit{cert}\}) \\
\quad \text{WHERE} \ \ \mathit{vstate} = \mathsf{V}\langle \ldots, \mathit{dag}, \ldots \rangle
\end{array}
$$

that updates the state of the author (if correct) of a new certificate, by adding the certificate to the DAG. This function is used below to express how the creation of a new certificate updates the state of the system.

We define a function

$$
\begin{array}{l}
\mathit{CreateNextEndor} : \mathit{Vstate} \times \mathit{Cert} \to \mathit{Vstate} \\
\mathit{CreateNextEndor}(\mathit{vstate}, \mathit{cert}) \triangleq \mathit{UpdateEar}(\mathit{vstate}, \mathit{ears} \cup \{\langle \mathit{auth}, \mathit{round} \rangle\}) \\
\quad \text{WHERE} \ \ \mathit{vstate} = \mathsf{V}\langle \ldots, \mathit{ears}, \ldots \rangle \\
\qquad \land \ \mathit{cert} = \mathsf{C}\langle \mathit{auth}, \mathit{round}, \ldots \rangle
\end{array}
$$

that updates the state of an endorser (if correct) of a new certificate, by adding the author and round to the set of endorsed pairs. This function is used below to express how the creation of a new certificate updates the state of the system.

We define a function

$$\mathit{CreateNext} : \mathit{State} \times \mathit{Cert} \to \mathit{State}$$

$$\begin{array}{l}
\mathit{CreateNext}(\mathit{state}, \mathit{cert}) \triangleq \mathsf{S}\langle \mathit{vstates}_p, \mathit{msgs} \cup \mathit{msgs}' \rangle \\
\quad \text{WHERE} \;\; \mathit{state} = \mathsf{S}\langle \mathit{vstates}, \mathit{msgs} \rangle \\
\qquad \wedge \;\; \mathit{cert} = \mathsf{C}\langle \mathit{auth}, \mathit{round}, \ldots, \mathit{endors} \rangle \\
\qquad \wedge \;\; \{\mathit{endor}_1, \ldots, \mathit{endor}_p\} = \{\mathit{endor} \in \mathit{endors} \mid \mathit{vstates}(\mathit{endor}) \neq \mathsf{faulty}\} \\
\qquad \wedge \;\; \mathit{vstates}_0 = \begin{cases} \mathit{vstates}\{\mathit{auth} \mapsto \mathit{CreateNextAuth}(\mathit{vstates}(\mathit{auth}), \mathit{cert})\} & \text{if} \;\; \mathit{vstates}(\mathit{auth}) \neq \mathsf{faulty} \\ \mathit{vstates} & \text{otherwise} \end{cases} \\
\qquad \wedge \;\; \forall i \in \{1, \ldots, p\} \,.\, \mathit{vstates}_i = \mathit{vstates}_{i-1}\{\mathit{endor}_i \mapsto \mathit{CreateNextEndor}(\mathit{vstates}(\mathit{endor}_i), \mathit{cert})\} \\
\qquad \wedge \;\; \mathit{msgs}' = \{\mathsf{M}\langle \mathit{cert}, \mathit{dest} \rangle \mid \mathit{dest} \in \mathcal{D}(\mathit{vstates}) \setminus \{\mathit{auth}\} \\
\qquad\qquad\qquad\qquad \wedge \; \mathit{vstates}(\mathit{dest}) \neq \mathsf{faulty}\}
\end{array}$$

that updates the state of the system when a new certificate is created:

1. If the author is correct, its state is updated according to *CreateNextAuth*, defined earlier.
2. The state of each correct endorser is updated according to *CreateNextEndor*, defined earlier.
3. The network is extended with messages containing the certificate, from the author to all the other validators if correct.

The above definition works as follows:

- We start with a system state *state*, consisting of a validator map *vstates* and network *msgs*, and with the new certificate *cert*.
- We take the set of correct endorsers, which in general is a subset of all endorsers, since some may be faulty.
- If the author is correct, we update its state, obtaining an updated validator map $\mathit{vstates}_0$. If the author is not correct, $\mathit{vstates}_0$ is *vstates*, unchanged. (We do this for uniformity, so that $\mathit{vstates}_0$ is the same starting point for updating the correct endorser states.)
- We update the state of each such correct endorser, one after the other, obtaining a sequence of validator maps $\mathit{vstates}_1$ through $\mathit{vstates}_p$, each updating the state of the corresponding correct endorser. (The order of the endorsers in the set enumeration is immaterial, as the validator map updates are independent. It is also immaterial whether there are repeated endorsers in the set enumeration, because the map updates are idempotent. The set of correct endorsers could be empty in principle, in which case $p = 0$, but that does not happen in a fault-tolerant system.)
- We create a set msgs' of new messages, each containing the certificate, and having all correct validators, except the author if correct, as destinations.
- The complete updated system state consists of the validator map $\mathit{vstates}_p$, and the network obtained by joining the old messages *msgs* with the new messages msgs'.

Any validator, correct or faulty, can author a new certificate, provided that it can get enough endorsers. If the author is faulty, obtaining enough endorsers is the only restriction. If the author is correct, it means that they are following the protocol, and thus there are additional restrictions, expressed in *CreateCond* (specifically, *CreateCondAuth*) that the author must satisfy.

Any validator, correct or faulty, can endorse a new certificate. If the endorser is faulty, there is no restriction. If the endorser is correct, it means that they are following the protocol, and thus there are additional restrictions, expressed in *CreateCond* (specifically, *CreateCondEndor*), that endorsers must satisfy.

The creation of the new messages msgs' models the first act of a reliable broadcast, in the precise technical sense used in the BFT literature. The reason why only messages to correct validators are created is that faulty validators can act arbitrarily regardless of the messages they receive. Their arbitrary behavior is only relevant to the extent that it may impact correct validators. In our model, the arbitrary behavior of faulty validators is limited to: authoring certificates at will, so long as, critically, they obtain enough signatures; and endorsing certificates at will, but not, critically, impersonating correct validators.

In our current model, a certificate creation event models, at a more abstract level, the Narwhal exchange of proposals, signatures, and certificates. Our certificate creation event represents the final act of those exchanges, after enough signatures have been collected.

2.5.2 Message Receipt

We define a predicate

$$\begin{array}{l} \mathit{ReceiveCond} : \mathit{State} \times \mathit{Msg} \to \mathbb{B} \\ \mathit{ReceiveCond}(\mathit{state}, \mathit{msg}) \triangleq \mathit{msg} \in \mathit{msgs} \\ \quad \text{WHERE } \mathit{state} = \mathsf{S}\langle \ldots, \mathit{msgs} \rangle \end{array}$$

that says whether a message can be received from the network: the message must be in the network.

We define a function

$$\begin{array}{l} \mathit{ReceiveNextDest} : \mathit{Vstate} \times \mathit{Cert} \to \mathit{Vstate} \\ \mathit{ReceiveNext}(\mathit{vstate}, \mathit{cert}) \triangleq \mathit{vstate}'' \\ \quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \ldots, \mathit{buf}, \mathit{ears}, \ldots \rangle \\ \qquad \wedge\ \mathit{vstate}' = \mathit{UpdateBuf}(\mathit{vstate}, \mathit{buf} \cup \{\mathit{cert}\}) \\ \qquad \wedge\ \mathit{vstate}'' = \mathit{UpdateEar}(\mathit{vstate}', \mathit{ears} \setminus \{\langle \mathit{auth}, \mathit{round} \rangle\}) \end{array}$$

that updates the state of the (correct) validator whose address is the destination of a message with a certificate. The certificate is added to the buffer. If the validator is an endorser of that certificate, the author and round is removed from the set of endorsed pairs, because now the validator has the whole certificate, and no longer needs the record of having endorsed it without having it; that set difference causes no change if the validator is not an endorser.

We define a function

$$\begin{array}{l} \mathit{ReceiveNext} : \mathit{State} \times \mathit{Msg} \to \mathit{State} \\ \mathit{ReceiveNext}(\mathit{state}, \mathit{msg}) \triangleq \mathsf{S}\langle \mathit{vstates}', \mathit{msgs} \setminus \{\mathit{msg}\} \rangle \\ \quad \text{WHERE } \mathit{state} = \mathsf{S}\langle \mathit{vstates}, \mathit{msgs} \rangle \\ \qquad \wedge\ \mathit{msg} = \mathsf{M}\langle \mathit{cert}, \mathit{dest} \rangle \\ \qquad \wedge\ \mathit{vstates}' = \mathit{vstates}\{\mathit{dest} \mapsto \mathit{ReceiveNext}(\mathit{vstates}(\mathit{dest}), \mathit{cert})\} \end{array}$$

that updates the state of the system when a message is received by a validator. It is an invariant, proved later, that the destination of a message in the network is always a correct validator, never a faulty one: messages are created when new certificates are created, according to the definition in Section 2.5.1, which only generates messages for correct validators. The predicate *ReceiveCond* defined earlier, upon which this *ReceiveNext* is conditioned, requires the message to be in the network, and therefore the destination address must be of a correct validator. The destination validator state is updated as defined in *ReceiveNextDest* defined earlier, and the message is removed from the network.

2.5.3 Certificate Storage

We define a predicate

$$\begin{array}{l} \mathit{StoreCondVal} : \mathit{Vstate} \times \mathit{Cert} \to \mathbb{B} \\ \mathit{StoreCondVal}(\mathit{vstate}, \mathit{cert}) \triangleq \mathit{cert} \in \mathit{buf} \\ \qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{HasPrevious}(\mathit{vstate}, \mathit{prevs}, \mathit{round}) \\ \quad \text{WHERE } \mathit{vstate} = \mathsf{V}\langle \ldots, \mathit{buf}, \ldots \rangle \\ \qquad \wedge\ \mathit{cert} = \mathsf{C}\langle \ldots, \mathit{round}, \ldots, \mathit{prevs}, \ldots \rangle \end{array}$$

that says whether a certificate can be stored in a validator's DAG from the standpoint of the validator state:

1. The certificate must be in the buffer of the validator.
2. The DAG must have all the certificates in the previous round referenced by the certificate.

Recall that it is an invariant (proved later) that *prevs* is empty if *round* is 1, and that *HasPrevious* also handles this special case.

We define a predicate

$$\begin{array}{l} \mathit{StoreCond} : \mathit{State} \times \mathit{Addr} \times \mathit{Cert} \to \mathbb{B} \\ \mathit{StoreCond}(\mathit{state}, \mathit{val}, \mathit{cert}) \triangleq \mathit{val} \in \mathcal{D}(\mathit{vstates}) \\ \qquad\qquad\qquad\qquad\qquad\ \wedge\ \mathit{vstates}(\mathit{val}) \neq \mathsf{faulty} \\ \qquad\qquad\qquad\qquad\qquad\ \wedge\ \mathit{StoreCondVal}(\mathit{vstates}(\mathit{val}), \mathit{cert}) \\ \quad \text{WHERE } \mathit{state} = \mathsf{S}\langle \mathit{vstates}, \ldots \rangle \end{array}$$

that says whether a certificate can be stored in a validator's DAG from the standpoint of the system state:

1. The validator in question must be a correct one.
2. The validator state must satisfy the appropriate conditions; see *StoreCondVal*, defined earlier.

We define a function

$$
\begin{array}{l}
\mathit{StoreNextVal} : \mathit{Vstate} \times \mathit{Cert} \to \mathit{Vstate} \\
\mathit{StoreNextVal}(\mathit{vstate}, \mathit{cert}) \triangleq \mathit{vstate}'' \\
\quad \text{WHERE} \;\; \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \mathit{buf}, \ldots \rangle \\
\qquad \wedge \;\; \mathit{cert} = \mathsf{C}\langle \ldots, \mathit{round}', \ldots \rangle \\
\qquad \wedge \;\; \mathit{vstate}' = \mathit{UpdateBuf}(\mathit{vstate}, \mathit{buf} \setminus \{\mathit{cert}\}) \\
\qquad \wedge \;\; \mathit{vstate}'' = \mathit{UpdateDag}(\mathit{vstate}', \mathit{dag} \cup \{\mathit{cert}\}) \\
\qquad \wedge \;\; \mathit{round}'' = \begin{cases} \mathit{round}' - 1 & \text{if} \;\; \mathit{round}' > \mathit{round} + 1 \\ \mathit{round} & \text{otherwise} \end{cases} \\
\qquad \wedge \;\; \mathit{vstate}''' = \mathit{UpdateRound}(\mathit{vstate}'', \mathit{round}'')
\end{array}
$$

that updates the state of a correct validator by storing a certificate from the buffer into the DAG:

1. The initial validator state *vstate* is updated by removing the certificate from the buffer.
2. The resulting validator state vstate' is updated by adding the certificate to the DAG.
3. The resulting validator state vstate'' is possibly updated by moving its current round to one less than the certificate, if that is ahead of the current round.
4. The resulting validator state vstate''' is the final one.

It is an invariant (proved later) that the DAG and buffer of each validator state are disjoint, and since this state change is conditioned under *StoreCondVal* defined earlier, The set subtraction and union operations indeed move the certificate from the buffer to the DAG. The round advancement happens to catch up with other validators, which must have moved to later rounds given that a certificate with a later round is received.

We define a function

$$
\begin{array}{l}
\mathit{StoreNext} : \mathit{State} \times \mathit{Addr} \times \mathit{Cert} \to \mathit{State} \\
\mathit{StoreNext}(\mathit{state}, \mathit{val}, \mathit{cert}) \triangleq \mathsf{S}\langle \mathit{vstates}', \mathit{msgs} \rangle \\
\quad \text{WHERE} \;\; \mathit{state} = \mathsf{S}\langle \mathit{vstates}, \mathit{msgs} \rangle \\
\qquad \wedge \;\; \mathit{vstates}' = \mathit{vstates}\{\mathit{val} \mapsto \mathit{StoreNextVal}(\mathit{vstates}(\mathit{val}), \mathit{cert})\}
\end{array}
$$

that updates the system state by storing a certificate from the buffer of a validator into the DAG of the validator. This is conditioned under *StoreCond* defined earlier, which ensures that *val* is a correct validator in the system.

This event moves a certificate from the buffer to the DAG, provided that all the certificates it is based on (i.e. the ones it references from the previous round) are already in the DAG. This is an important condition, which guarantees a number of properties (proved later) of the DAGs of correct validators, such as the closure of certificates' causal histories. The purpose of the buffer, as explained in Section 2.1.6, is to hold certificates received via messages until the previous certificates are in the DAG, given that certificates may be received in any order.

2.5.4 Round Advancement

We define a predicate

$$
\begin{array}{l}
\mathit{AdvanceCondValEven} : \mathit{Vstate} \times \mathcal{P}_\omega(\mathit{Addr}) \times (\mathbb{N} \setminus \{0\}) \to \mathbb{B} \\
\mathit{AdvanceCondValEven}(\mathit{vstate}, \mathit{vals}, \mathit{quor}) \;\triangleq\; \mathsf{C}\langle \mathit{lead}, \mathit{round}, \ldots \rangle \in \mathit{dag} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \;\; \mathit{timer} = \mathsf{expired} \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; |\{\mathit{cert} \in \mathit{dag} \mid \mathit{cert} = \mathsf{C}\langle \ldots, \mathit{round}, \ldots \rangle\}| \geq \mathit{quor} \\
\quad \text{WHERE} \;\; \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \ldots, \mathit{timer} \rangle \\
\qquad \wedge \;\; \mathit{lead} = \mathit{Leader}(\mathit{vals}, \mathit{round})
\end{array}
$$

that says when an even round number can be incremented by one in the state of a validator:

1. We choose the address of the leader of the round.

2. The round can be incremented if one of the following conditions holds:
   (a) The DAG has the anchor for the round, i.e. the certificate with the leader as author at that round.
   (b) The timer has expired and the round already has as many certificates as the quorum number.

This predicate is used, below, with $vals = \mathcal{D}(vstates)$ and $quor = Quorum(state)$, and when *round* is even.

Normally, when a validator is in an even round, they wait for the anchor at that round, which is the certificate with that round and with the leader as author (the author may be that or another validator). When the anchor is received (or authored if the validator is the leader), the validator can advance to the next round. However, if the leader is faulty, the anchor may never arrive (it may not be generated at all). Thus, if there are already a number of certificates at least equal to the quorum, and the timer has expired, the validator should not wait for the anchor, which may never arrive, because the remaining validators whose certificates have not arrived yet (including the anchor) may be all faulty.

We define a function

$$
\begin{array}{l}
Tally : \mathcal{P}_\omega(Cert) \times Round \times Addr \to \mathbb{N} \times \mathbb{N} \\
Tally(dag, round, lead) \triangleq \langle yes, no \rangle \\
\quad \text{WHERE} \;\; yes = |\{\mathsf{C}\langle \ldots, round, \ldots, prevs, \ldots \rangle \in dag \mid lead \in prevs\}| \\
\qquad \wedge \;\; no = |\{\mathsf{C}\langle \ldots, round, \ldots, prevs, \ldots \rangle \in dag \mid lead \notin prevs\}|
\end{array}
$$

that tallies the positive and negative votes for a leader address *lead* in a round *round*, within a DAG *dag*. When this function is used, *round* is an odd round number different from one, and *lead* is the address of the leader for the even round just before *round*. We consider all the certificates in the DAG at round *round*: if a certificate has an edge pointing to the leader, it counts as a positive vote; otherwise, it counts as a negative vote. Only certificates in the DAG are considered; the absence of a certificate at round *round* for a certain author does not count as either a positive vote or a negative vote.

We define a predicate

$$
\begin{array}{l}
AdvanceCondValOdd : Vstate \times \mathcal{P}_\omega(Addr) \times (\mathbb{N} \setminus \{0\}) \times \mathbb{N} \to \mathbb{B} \\
AdvanceCondValOdd(vstate, vals, quor, maxf) \;\triangleq\; \mathsf{C}\langle lead, round - 1, \ldots \rangle \notin dag \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \;\; yes \geq maxf + 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \;\; no \geq quor \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \;\; timer = \mathsf{expired} \\
\quad \text{WHERE} \;\; vstate = \mathsf{V}\langle round, dag, \ldots, timer \rangle \\
\qquad \wedge \;\; lead = Leader(vals, round - 1) \\
\qquad \wedge \;\; \langle yes, no \rangle = Tally(dag, round, lead)
\end{array}
$$

that says when an odd non-one round number can be incremented by one in the state of a validator:

1. We choose the address of the leader of the immediately preceding even round.
2. The round can be incremented if one of the following conditions holds:
   (a) The anchor at the even round is absent.
   (b) The odd round has at least as many positive votes for the even-round leader as one more than the maximum number of faulty validators.
   (c) The odd round has at least as many negative votes for the odd-round leader as the quorum number.
   (d) The timer has expired.

This predicate is used, below, with $vals = \mathcal{D}(vstates)$, $quor = Quorum(state)$, and $maxf = MaxFaulty(state)$, and when *round* is odd and not one.

The condition on positive votes being at least one more than the maximum number of faulty validators is an important one: by reasoning about the intersection between these positive voters and the previous certificates at the same odd round that a certificate in the subsequent even round links to, we can guarantee consistency in the anchors in the DAG of different correct validators, and thus guarantee that the blockchains of different correct validators never fork. All of this is proved later.

We define a predicate

$$
\begin{array}{l}
AdvanceCondVal : Vstate \times \mathcal{P}_\omega(Addr) \times (\mathbb{N} \setminus \{0\}) \times \mathbb{N} \to \mathbb{B} \\
AdvanceCondVal(vstate, vals, quor, maxf) \triangleq \begin{array}{ll}
 & round \bmod 2 = 0 \land \\
 & AdvanceCondValEven(vstate, vals, quor) \\
 & round \bmod 2 = 1 \land \\
\lor & round \neq 1 \land \\
 & AdvanceCondValOdd(vstate, vals, quor, maxf) \\
\lor & round = 1
\end{array} \\
\quad \text{WHERE } vstate = \mathsf{V}\langle \ldots, round, \ldots \rangle
\end{array}
$$

that says when the round number of a validator can be incremented from the standpoint of the state of the validator, which is the case if one of the following conditions holds:

1. The round number is even, and *AdvanceCondValEven* (defined earlier) holds.
2. The round number is odd but not 1, and *AdvanceCondValOdd* (defined earlier) holds.
3. The round number is 1.

We define a predicate

$$
\begin{array}{l}
AdvanceCond : State \times Addr \to \mathbb{B} \\
AdvanceCond(state, val) \triangleq \begin{array}{ll}
 & val \in \mathcal{D}(vstates) \\
\land & vstates(val) \neq \mathsf{faulty} \\
\land & AdvanceCondVal(vstates(val), vals, quor, maxf)
\end{array} \\
\quad \text{WHERE } state = \mathsf{S}\langle vstates, \ldots \rangle \\
\qquad \land\ vals = \mathcal{D}(vstates) \\
\qquad \land\ quor = Quorum(state) \\
\qquad \land\ maxf = MaxFaulty(state)
\end{array}
$$

that says when the round number of a validator in the system can be incremented:

1. The validator in question must be a correct one.
2. The validator state must satisfy the appropriate conditions; see *AdvanceCondVal*, defined earlier.

We define a function

$$
\begin{array}{l}
AdvanceNextVal : Vstate \to Vstate \\
AdvanceNextVal(vstate) \triangleq vstate'' \\
\quad \text{WHERE } vstate = \mathsf{V}\langle round, \ldots, timer \rangle \\
\qquad \land\ vstate' = UpdateRound(vstate, round + 1) \\
\qquad \land\ vstate'' = UpdateTimer(vstate', \mathsf{running})
\end{array}
$$

that advances the round number of a validator:

1. The initial validator state *vstate* is updated by incrementing the round by one.
2. The resulting validator state $vstate'$ is updated by starting the timer, i.e. setting it to running.
3. The resulting validator state $vstate''$ is the final one.

We define a function

$$
\begin{array}{l}
AdvanceNext : State \times Addr \to State \\
AdvanceNext(state, val) \triangleq \mathsf{S}\langle vstates', msgs \rangle \\
\quad \text{WHERE } state = \mathsf{S}\langle vstates, msgs \rangle \\
\qquad \land\ vstates' = vstates\{val \mapsto AdvanceNextVal(vstates(val))\}
\end{array}
$$

that updates the system state by advancing the round of a correct validator. This is conditioned under *AdvanceCond* defined earlier, which ensures that *val* is a correct validator in the system.

2.5.5 Certificate Commitment

We define a predicate

$$
\begin{array}{l}
\mathit{CommitCondVal} : \mathit{Vstate} \times \mathcal{P}_\omega(\mathit{Addr}) \times \mathbb{N} \to \mathbb{B} \\
\mathit{CommitCondVal}(\mathit{vstate}, \mathit{vals}, \mathit{maxf}) \triangleq \mathit{round} \bmod 2 = 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{round} \neq 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{round} - 1 > \mathit{last} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathsf{C}\langle \mathit{lead}, \mathit{round} - 1, \ldots \rangle \in \mathit{dag} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{yes} \geq \mathit{maxf} + 1 \\
\quad \text{WHERE}\ \ \mathit{vstate} = \mathsf{V}\langle \mathit{round}, \mathit{dag}, \ldots, \mathit{last}, \ldots \rangle \\
\qquad\quad \wedge\ \mathit{lead} = \mathit{Leader}(\mathit{vals}, \mathit{round} - 1) \\
\qquad\quad \wedge\ \langle \mathit{yes}, \ldots \rangle = \mathit{Tally}(\mathit{dag}, \mathit{round}, \mathit{lead})
\end{array}
$$

that says whether a validator can commit one or more anchors from the standpoint of the validator state:

1. The validator must be in an odd round.
2. The validator must not be in round 1.
3. The immediately preceding even round must have the anchor, i.e. the certificate authored by the leader of that round.
4. The current odd round must have a number of positive votes for the leader that is at least one more than the maximum number of faulty validators.

This predicate is used, below, with $\mathit{vals} = \mathcal{D}(\mathit{vstates})$ and $\mathit{maxf} = \mathit{MaxFaulty}(\mathit{state})$.

We define a predicate

$$
\begin{array}{l}
\mathit{CommitCond} : \mathit{State} \times \mathit{Addr} \to \mathbb{B} \\
\mathit{CommitCond}(\mathit{state}, \mathit{val}) \triangleq \mathit{val} \in \mathcal{D}(\mathit{vstates}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{vstates}(\mathit{val}) \neq \mathsf{faulty} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{CommitCondVal}(\mathit{vstates}(\mathit{val}), \mathit{vals}, \mathit{maxf}) \\
\quad \text{WHERE}\ \ \mathit{state} = \mathsf{S}\langle \mathit{vstates}, \ldots \rangle \\
\qquad\quad \wedge\ \mathit{vals} = \mathcal{D}(\mathit{vstates}) \\
\qquad\quad \wedge\ \mathit{maxf} = \mathit{MaxFaulty}(\mathit{state})
\end{array}
$$

that says whether a validator in the system can commit one or more anchors:

1. The validator in question must be a correct one.
2. The validator must satisfy the appropriate conditions; see *CommitCondVal*, defined earlier.

We define a function

$$
\begin{array}{l}
\mathit{AnchorsToCommit} : \mathbb{N} \times \mathit{Round} \times \mathcal{P}_\omega(\mathit{Cert}) \times \mathcal{P}_\omega(\mathit{Addr}) \to \mathit{Cert}^* \\
\mathit{AnchorsToCommit}(\mathit{last}, \mathit{round}, \mathit{dag}, \mathit{vals}) \triangleq [\mathit{cert}_1, \ldots, \mathit{cert}_n] \\
\quad \text{WHERE}\ \ [x_0, \ldots, x_n] \in \{x \in \mathbb{N} \mid x \bmod 2 = 0\}^+ \\
\qquad \wedge\ n \geq 1 \\
\qquad \wedge\ \mathit{last} = x_0 < \ldots < x_n = \mathit{round} \\
\qquad \wedge\ \forall i \in \{1, \ldots, n\}\ .\ x_{i-1} = \max\{x \in \mathbb{N} \mid x < x_i \wedge (x = 0 \vee \mathit{AnchorPath}(x_i, x, \mathit{dag}, \mathit{vals}))\} \\
\qquad \wedge\ \forall i \in \{1, \ldots, n\}\ .\ \mathit{cert}_i = \mathsf{C}\langle \ldots, x_i, \ldots \rangle \in \mathit{Anchors}(\mathit{dag}, \mathit{vals})
\end{array}
$$

that yields the anchors to commit when the anchor at the even round *round* has sufficient votes in the immediately subsequent odd round (as expressed by *CommitCondVal*). We start with round $x_n = \mathit{round}$ and the anchor cert_n at that round. We find the closest (i.e. largest) round $x_{n-1} < x_n$ with an anchor cert_{n-1} such that there is a path from cert_n to cert_{n-1}. This may skip rounds between x_{n-1} and x_n, if those rounds lack the anchor or lack a path to their anchor from cert_n. We repeat the process to find the closest anchor cert_{n-2} to which there is a path from cert_{n-1}, again possibly skipping rounds in between. We continue like this until we reach $x_0 = \mathit{last}$, where *last* is the last committed round in the validator state, or 0 if the validator has not committed any round yet. Note that the possibility of $x_0 = \mathit{last} = 0$ is catered for by the definition above, via the disjunct $x = 0$ in the set whose maximum is taken. The function just defined yields the anchors to commit in chronological order, from oldest to newest. There is always at least one

anchor, under the predicate *CommitCondVal* that conditions this function; there could be just one anchor, or more than one anchor, with $cert_1$ having a path to the last committed one, or to none if $last = 0$.

We define a function

$$
\begin{array}{l}
AddBlock : Block^* \times \mathcal{P}_\omega(Cert) \times Cert \times \mathcal{P}_\omega(Cert) \to Block^* \times \mathcal{P}_\omega(Cert) \\
AddBlock(blocks, comm, cert, dag) \triangleq \langle blocks', comm' \rangle \\
\quad \text{WHERE } cert = \mathsf{C}\langle \ldots, round, \ldots \rangle \\
\qquad \wedge\ [cert_1, \ldots, cert_m] = Order(\{cert' \in dag \setminus comm \mid Path(cert, cert', dag)\}) \\
\qquad \wedge\ \forall i \in \{1, \ldots, m\}\ .\ cert_i = \mathsf{C}\langle \ldots, \overline{trans}_i, \ldots \rangle \\
\qquad \wedge\ \overline{trans} = \overline{trans}_1 \bowtie \cdots \bowtie \overline{trans}_m \\
\qquad \wedge\ blocks' = blocks \bowtie [\mathsf{B}\langle \overline{trans}, round \rangle] \\
\qquad \wedge\ comm' = comm \cup \{cert_1, \ldots, cert_m\}
\end{array}
$$

that adds a block to the blockchain in a validator state. Here *blocks*, *comm*, and *dag* are components of the validator state, and *cert* is an anchor to be committed, whose round is *round*. We take the causal history of *cert*, i.e. all the certificates in the DAG to which *cert* has a path to, but we exclude the certificates already committed (the ones in *comm*). We order those certificates in some way (the same for all validators; see Section 2.3.13), and we concatenate all the transactions from those certificates, in order. We form a block with all those transactions $\overline{trans}$, and with round *round*, and we return the blockchain extended with that block. We also return the updated set of committed certificates.

We define a function

$$
\begin{array}{l}
commitNextVal : Vstate \times \mathcal{P}_\omega(Addr) \to Vstate \\
commitNextVal(vstate, vals) \triangleq vstate'' \\
\quad \text{WHERE } vstate = \mathsf{V}\langle round, dag, \ldots, last, blocks, comm, \ldots \rangle \\
\qquad \wedge\ [cert_1, \ldots, cert_n] = AnchorsToCommit(last, round, dag, vals) \\
\qquad \wedge\ blocks_0 = blocks \\
\qquad \wedge\ comm_0 = comm \\
\qquad \wedge\ \forall i \in \{1, \ldots, n\}\ .\ \langle blocks_i, comm_i \rangle = AddBlock(blocks_{i-1}, comm_{i-1}, cert_i, dag) \\
\qquad \wedge\ vstate' = UpdateBlocks(vstate, blocks_n) \\
\qquad \wedge\ vstate'' = UpdateComm(vstate', comm_n)
\end{array}
$$

that updates the state of a validator by committing one or more anchors. When this is used (below), *vals* is the set of all validator addresses. This predicate is conditioned under *CommitCondVal*, defined earlier, which means that there are one or more anchors to commit, which we obtain from *AnchorsToCommit*. We add a block for each anchor, from oldest to newest, and we update the validator state with the final blockchain and the final set of committed certificates.

We define a function

$$
\begin{array}{l}
CommitNext : State \times Addr \to State \\
CommitNext(state, val) \triangleq \mathsf{S}\langle vstates', msgs \rangle \\
\quad \text{WHERE } state = \mathsf{S}\langle vstates, msgs \rangle \\
\qquad \wedge\ vstates' = vstates\{val \mapsto commitNextVal(vstates(val), \mathcal{D}(vstates))\}
\end{array}
$$

that updates the system state by committing anchors in a correct validator. This is conditioned under *CommitCond* defined earlier, which ensures that the validator is correct and has anchors to commit.

2.5.6 Timer Expiration

We define a predicate

$$
\begin{array}{l}
TimeoutCond : State \times Addr \to \mathbb{B} \\
TimeoutCond(state, val) \triangleq val \in \mathcal{D}(vstates) \\
\qquad\qquad\qquad\qquad \wedge\ vstates(val) = \mathsf{V}\langle \ldots, timer \rangle \neq \mathsf{faulty} \\
\qquad\qquad\qquad\qquad \wedge\ timer = \mathsf{running} \\
\quad \text{WHERE } state = \mathsf{S}\langle vstates, \ldots \rangle
\end{array}
$$

that says when a timeout event in a validator in the system. The validator in question must be a correct one, and its timer must be running.

We define a function

$$
\begin{array}{l}
\mathit{TimeoutNext} : \mathit{State} \times \mathit{Addr} \to \mathit{State} \\
\\
\mathit{TimeoutNext}(\mathit{state}, \mathit{val}) \triangleq \mathsf{S}\langle \mathit{vstates}', \mathit{msgs} \rangle \\
\quad \text{WHERE} \;\; \mathit{state} = \mathsf{S}\langle \mathit{vstates}, \mathit{msgs} \rangle \\
\qquad \wedge \;\; \mathit{vstate} = \mathit{vstates}(\mathit{val}) \\
\qquad \wedge \;\; \mathit{vstate}' = \mathit{UpdateTimer}(\mathit{vstate}, \mathsf{expired}) \\
\qquad \wedge \;\; \mathit{vstates}' = \mathit{vstates}\{\mathit{val} \mapsto \mathit{vstate}'\}
\end{array}
$$

that updates the state of a validator in the system when a timeout happens. This is conditioned under *TimeoutCond* defined earlier, which guarantees that *val* is the address of a correct validator. The timer state component of the validator is set to indicate expiration.

Our formal specification does not model real time explicitly, but it models the presence of the timers (one per correct validator) and the possible occurrence of timeouts. This enables reasoning about the behavior of the system when timeouts occur and do not occur, based on sequences of events devoid of time information.

2.5.7 Transition Predicate and Function

We define the *transition predicate* as

$$
\begin{array}{l}
\mathit{EventCond} : \mathit{State} \times \mathit{Event} \to \mathbb{B} \\
\\
\mathit{EventCond}(\mathit{state}, \mathsf{Ecreate}[\mathit{cert}]) \triangleq \mathit{CreateCond}(\mathit{state}, \mathit{cert}) \\
\mathit{EventCond}(\mathit{state}, \mathsf{Ereceive}[\mathit{msg}]) \triangleq \mathit{ReceiveCond}(\mathit{state}, \mathit{msg}) \\
\mathit{EventCond}(\mathit{state}, \mathsf{Estore}[\mathit{cert}, \mathit{val}]) \triangleq \mathit{StoreCond}(\mathit{state}, \mathit{cert}, \mathit{val}) \\
\mathit{EventCond}(\mathit{state}, \mathsf{Eadvance}[\mathit{val}]) \triangleq \mathit{AdvanceCond}(\mathit{state}, \mathit{val}) \\
\mathit{EventCond}(\mathit{state}, \mathsf{Ecommit}[\mathit{val}]) \triangleq \mathit{CommitCond}(\mathit{state}, \mathit{val}) \\
\mathit{EventCond}(\mathit{state}, \mathsf{Etimer}[\mathit{val}]) \triangleq \mathit{TimeoutCond}(\mathit{state}, \mathit{val})
\end{array}
$$

i.e. in terms of the predicates defined earlier, based on the kind of event.

We define the *transition function* as

$$
\begin{array}{l}
\mathit{EventNext} : \mathit{State} \times \mathit{Event} \to \mathit{State} \\
\\
\mathit{EventNext}(\mathit{state}, \mathsf{Ecreate}[\mathit{cert}]) \triangleq \mathit{CreateNext}(\mathit{state}, \mathit{cert}) \\
\mathit{EventNext}(\mathit{state}, \mathsf{Ereceive}[\mathit{msg}]) \triangleq \mathit{ReceiveNext}(\mathit{state}, \mathit{msg}) \\
\mathit{EventNext}(\mathit{state}, \mathsf{Estore}[\mathit{cert}, \mathit{val}]) \triangleq \mathit{StoreNext}(\mathit{state}, \mathit{cert}, \mathit{val}) \\
\mathit{EventNext}(\mathit{state}, \mathsf{Eadvance}[\mathit{val}]) \triangleq \mathit{AdvanceNext}(\mathit{state}, \mathit{val}) \\
\mathit{EventNext}(\mathit{state}, \mathsf{Ecommit}[\mathit{val}]) \triangleq \mathit{CommitNext}(\mathit{state}, \mathit{val}) \\
\mathit{EventNext}(\mathit{state}, \mathsf{Etimer}[\mathit{val}]) \triangleq \mathit{TimeoutNext}(\mathit{state}, \mathit{val})
\end{array}
$$

i.e. in terms of the functions defined earlier, based on the kind of event.

2.6 Labeled State Transition System

We define the *AleoBFT labeled state transition system* as

$$\mathit{Sys}_{\text{AleoBFT}} = \langle S_{\text{AleoBFT}}, E_{\text{AleoBFT}}, I_{\text{AleoBFT}}, T_{\text{AleoBFT}} \rangle$$

where

$$
\begin{array}{l}
S_{\text{AleoBFT}} \triangleq \mathit{State} \\
E_{\text{AleoBFT}} \triangleq \mathit{Event} \\
I_{\text{AleoBFT}} \triangleq \mathit{InitState} \\
T_{\text{AleoBFT}} \triangleq \{\langle \mathit{state}, \mathit{event}, \mathit{state}' \rangle \in \mathit{State} \times \mathit{Event} \times \mathit{State} \mid \mathit{EventCond}(\mathit{state}, \mathit{event}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \; \mathit{EventNext}(\mathit{state}, \mathit{event}) = \mathit{state}'\}
\end{array}
$$

i.e. the set of possible states is defined in Section 2.1.8, the set of possible events is defined in Section 2.2, the set of possible initial states is defined in Section 2.4, and the transition relation is determined by the predicate and function defined in Section 2.5.7, as the set of triples of old states, events, and new states such that the predicate holds on the old state and event and the function yields the new state on the old state and event.

This AleoBFT labeled state transition system is an instance of the generic notion of labeled state transition system introduced at the beginning of Section 2.

3 Correctness

This section will formulate theorems, and sketch their proofs, for some correctness properties of AleoBFT.

4 Conclusion

This section will draw some closing remarks, and describe future work.

A Mathematical Notation

A.1 Booleans and Numbers

Booleans $\mathbb{B}$ is the set of booleans, $\{True, False\}$.

Natural Numbers $\mathbb{N}$ is the set of natural numbers, $\{0, 1, 2, \ldots\}$.

A.2 Logic

Definitional Equality $\triangleq$ is used to define a new term from known terms. For example, $Round \triangleq \mathbb{N} \setminus \{0\}$ defines *Round* as the set of positive integers.

Conjunction $A \wedge B$ is the conjunction of the assertions A and B. It is true exactly when both A and B are true.

Disjunction $A \vee B$ is the disjunction of the assertions A and B. It is true exactly when either A or B or both are true.

Implication $A \Longrightarrow B$ is equivalent to "if A then B". For example,

$$i \neq i' \Longrightarrow cert_i \neq cert_{i'}$$

states that if i is not equal to i', then $cert_i$ is not equal to $cert_{i'}$.

Universal Quantification $\forall a \,.\, P$ states that *for all* a, the predicate P is true. Usually the variable of quantification has a restriction, so $\forall a \in A \,.\, P$ is equivalent to $\forall a \,.\, a \in A \Longrightarrow P$. For example, the following states that all values of i from 1 to m are less than $m+1$: $\forall i \in \{1, \ldots, m\} \,.\, i < m+1$. This can also be read as "for all i, if i is in the set of integers from 1 to m, then i is less than $m+1$." Multiple universally-quantified variables can be used with a single forall symbol: $\forall\, a,\, b.\, P$ is equivalent to $\forall a \,.\, \forall b \,.\, P$.

Existential Quantification $\exists a \,.\, P$ states that *there exists* an a such that the predicate P is true. Usually the the variable of quantification has a restriction: $\exists a \in A \,.\, P$ is equivalent to $\exists a \,.\, a \in A \wedge P$. The quantified form can be a constructor with multiple existentially-quantified variables. For example,

$$\exists [cert_1, \ldots, cert_p] \in Cert^+ \,.\, \langle \textit{predicate involving the } cert_i \textit{ variables} \rangle$$

states that "there exists a sequence of certificates in $Cert^+$, which we will call $cert_1$ through $cert_p$, such that ...". By convention, if $p = 1$, then $[cert_1, \ldots, cert_p]$ means $[cert_1]$.

A.3 Sets

Empty Set $\emptyset$ is the empty set.

Set Comprehension $\{a \mid P\}$ is the set of all elements a such that the predicate P is true. There can be an expression to the left of the vertical bar, which is evaluated for all values of the variables in the expression that satisfy the predicate, and the results of evaluation collected in the set. For example,

$$\{\mathsf{B}\langle\overline{trans}, round\rangle \mid \overline{trans} \in Trans^* \ \wedge \ round \in Round\}$$

defines the set of blocks that can be formed from a transaction sequence and a round number.

If there is a membership test on the left side of the vertical bar, then the set is restricted to those elements that pass the test. I.e. $\{a \in A \mid P\}$ is equivalent to $\{a \mid a \in A \ \wedge \ P\}$. For example,

$$\{val \in \mathcal{D}(vstates) \mid vstates(val) = \mathsf{faulty}\}$$

defines the set of faulty validators.

Cardinality $|A|$ is the cardinality of the finite set A, i.e. the number of elements A contains.

Set Membership $a \in A$ states that a is an element of the set A.

Set Difference $A \setminus B$ is the set of elements in A that are not in B. For example, $\mathbb{N} \setminus \{0\}$ is the set of positive integers, $\{1, 2, 3, \ldots\}$.

Set Union $A \cup B$ is the union of the sets A and B. For example, $Round \cup \{0\}$ is the set containing the round numbers and 0, i.e. $\mathbb{N}$.

Inclusive Subset $A \subseteq B$ states that A is a subset of B or equal to B.

Finite Powerset $\mathcal{P}_\omega(A)$ is the finite powerset of the set A, i.e. the set of all finite subsets of A. Typically this is used along with set membership. For example, $prevs \in \mathcal{P}_\omega(Addr)$ states that *prevs* is a finite set of addresses. This idiom is used instead of $prevs \subseteq Addr$ because the the latter does not restrict *prevs* to being finite.

Cartesian Product $A \times B$ is the Cartesian product of the sets A and B, i.e. the set of all pairs (2-tuples) $\langle a, b\rangle$ where a is an element of A and b is an element of B. For example, $Addr \times Round$ is the set of all pairs $\langle auth, round\rangle$ where *auth* is an address in *Addr* and *round* is a round number in *Round*. Cartesian product can operate on more than two sets, in which case the tuples have more than two elements.

A.4 Sequences

Empty Sequence ϵ is the empty sequence.

Sequence Constructor $[a_1, \ldots, a_n]$ is a sequence of elements a_1 through a_n. By convention, if $n = 1$ then $[a_1, \ldots, a_n]$ is $[a_1]$.

Sequence Indicator $\overline{a}$ is a sequence, which is indicated by the line over it. For example, $\overline{trans} \in Trans^*$ states that $\overline{trans}$ is a member of $Trans^*$. Described another way, after considering the next paragraph, $\overline{trans} \in Trans^*$ states that $\overline{trans}$ is a sequence, possibly empty, of transactions taken from the set *Trans*.

Set of Sequences over a Set A^* is the set of all finite sequences of elements of the set A. For example, $Trans^*$ is the set of all finite sequences of transactions, given that *Trans* is the set of all transactions.

Set of Nonempty Sequences over a Set A^+ is the set of all non-empty finite sequences of elements of the set A.

Sequence Concatenation $\overline{a} \bowtie \overline{b}$ is the sequence obtained by concatenating the sequences $\overline{a}$ and $\overline{b}$.

A.5 Tuples

Tuple Constructor A tuple is constructed with one or more elements within angle brackets. Examples: $\langle auth, round\rangle$; $\langle S, E, I, T\rangle$.

Named Tuple Constructor $A(..)$, where A is an explicitly-defined name in sans-serif font, is a function-call-like constructor of a "colored" tuple where the color is the name of the object constructed. For example, $\mathsf{B}\langle\overline{trans}, round\rangle$ is a tuple with name B, which names a block, and two elements: a transaction sequence and a round number. It is equivalent to $\langle\mathsf{B}, \overline{trans}, round\rangle$.

For reference, the most common named tuples are:

- B: block
- C: certificate
- M: message
- S: system state
- V: validator state

A.6 Maps

A map is another name for a finite set of ordered pairs $\langle a, b\rangle$, where a is the key and b is the value. The key is unique in the map. The domain is the set of keys and the range is the set of values. A map is also a mathematical function.

Set of Maps $A \rightsquigarrow B$ is the set of finite maps that have a domain in $\mathcal{P}_\omega(A)$ and a range in $\mathcal{P}_\omega(B)$. For example, $Addr \rightsquigarrow Vstate \cup \{\mathsf{faulty}\}$ is the set of maps whose domain is a finite set of addresses and whose range is an finite subset of $Vstate \cup \{\mathsf{faulty}\}$.

Map Declaration $m : A \rightsquigarrow B$ states that m is a map whose domain is a finite inclusive subset of A (which could be equal to A if A is finite) and whose range is a finite inclusive subset of B. For example, to state that a map *vstates* is in the set of maps mentioned above, we write

$$vstates : Addr \rightsquigarrow Vstate \cup \{\mathsf{faulty}\}$$

I.e. *vstates* is a map whose domain is a finite set of addresses and whose range is a finite inclusive subset of $Vstate \cup \{\mathsf{faulty}\}$.

Domain $\mathcal{D}(A)$ is the domain of map A. For example, $\mathcal{D}(vstates)$ is the domain of the map *vstates*.

Map Update $m\{a \mapsto b\}$ is the map formed by starting with m and changing key a's value to b. For example, $vstates\{val \mapsto newstate\}$ is the map formed by starting with *vstates* and updating key *val*'s value to *newstate*.

A.7 Functions

A function is another name for a set of ordered pairs $\langle a, b\rangle$, where a is the input and b is the output. The input is unique in the set, i.e. for a given input, there is only one possible output. The domain is the set of inputs permitted, which can be an infinite set, and the range is the possibly infinite set of possible outputs.

Set of Functions $A \to B$ is the set of functions whose domain is A and whose range is a subset-or-equal of B. Each function is *total*, i.e. it is defined for all elements of A. For example, $State \to \mathbb{N} \setminus \{0\}$ is the set of functions whose domain is *State* and whose range is a subset-or-equal of $\mathbb{N} \setminus \{0\}$.

Function Declaration $f : A \to B$ states that f is a function whose domain is A and whose range is a subset-or-equal of B. For example, to state that a function *NumAll* is in the set of functions mentioned above, we write $NumAll : State \to \mathbb{N} \setminus \{0\}$. I.e. *NumAll* is a function whose input can be any element of *State* and whose output is a positive integer.

References

[APa] Aleo Team and Provable Team. Aleo. `https://aleo.org`.

[APb] Aleo Team and Provable Team. snarkOS. `https://github.com/AleoNet/snarkOS`.

[APc] Aleo Team and Provable Team. snarkVM. `https://github.com/AleoNet/snarkVM`.

[CM] Alessandro Coglio and Eric McCarthy. AleoBFT formal specification and proofs in ACL2. `https://github.com/ProvableHQ/aleo-acl2/tree/master/bft`.

[DKKSS22] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus. In *Proc. 17th European Conference on Computer Systems (EuroSys)*, pages 34–50, 2022.

[SGSKK22a] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *Proc. 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2705–2718, 2022.

[SGSKK22b] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: The partially synchronous version, 2022.