Formal Verification of Blockchain Nonforking in DAG-Based BFT Consensus with Dynamic Stake

Alessandro Coglio Eric McCarthy
Provable Inc.

April 23, 2025

Abstract

Blockchain consensus protocols enable participants to agree on consistent views of the blockchain that may be ahead or behind relative to each other but do not fork into different chains. A number of recently popular Byzantine-fault-tolerant (BFT) protocols first construct a directed acyclic graph (DAG) that partially orders transactions, then linearize the DAG into a blockchain that totally orders transactions. The definitions and correctness proofs of these DAG-based protocols typically assume that the set of participants is fixed, which is impractical in long-lived blockchains. Additionally, only a few of those proofs have been machine-checked, uncovering errors in some published proofs. We developed a formal model of a DAG-based BFT protocol with dynamic stake, where participants can join and leave at every block, with stake used to weigh decisions in the protocol. We formally proved that blockchains never fork in the model, also clarifying how BFT bounds on faulty participants generalize to these highly dynamic sets of participants. Our model and proofs are formalized in the ACL2 theorem prover, apply to arbitrarily long executions and arbitrarily large system states, and are verified in 1 minute by ACL2.

1 Introduction

A *blockchain* is an append-only ledger of *transactions* (transfers of value, smart contract executions, etc.), organized in *blocks*, sequentially linked as a chain. A blockchain *consensus protocol* enables its decentralized participants, called *validators*, to settle on consistent views of the blockchain. Consistency means that different validators' blockchains do not *fork*: one may be ahead of the other by one or more blocks, but they must not append different blocks to a common prefix. Some protocols actually allow temporary forks [31], while others, such as those described next, do not.

In a recently popular family of *DAG-based* protocols [8, 18, 13, 32, 20, 35, 36, 23, 19, 39, 40, 30, 24, 38, 5, 17, 16, 7, 37, 21, 41, 29], validators exchange messages to construct a directed acyclic graph (DAG) that partially orders batches of transactions, which validators individually linearize, without additional messages, into a blockchain that totally orders transactions.[1] Once a validator settles on a prefix of blocks, it can no longer be modified: blockchains never fork, even temporarily. There are two sub-protocols: one to construct the DAG, and one to construct the blockchain, with the latter layered on top of the former. These protocols are designed to be *Byzantine-fault-tolerant* (*BFT*): they work in spite of a bounded subset of validators exhibiting arbitrary behavior (short of unreasonable abilities like breaking cryptography). Specifically, the protocols satisfy blockchain nonforking and other properties of interest, so long as the number of faulty validators is less than one third of the total.

The literature referenced above provides proofs of blockchain nonforking and other properties of the protocols, but those proofs are typically not very detailed. Only a few of them have been formalized and machine-checked [9, 15], which uncovered errors in some of the originally published proofs [15].

The definitions and proofs of the protocols in the literature referenced above typically assume that the set of validators is fixed. But in a long-lived blockchain, a fixed validator set is impractical. Thus, extensions to the above protocols have been developed, where the validator set can change *dynamically*. A particularly dynamic approach is taken in the Aleo blockchain [1], whose consensus protocol AleoBFT [2, 3] allows the

[1] Some protocols do not actually build blockchains, but still linearize transactions, which involves the same issues.

validator set to potentially change at every block, via transactions that *bond* and *unbond* validators with *stake* that is used to weigh decisions in the protocol. This prevents a layered composition of sub-protocols and proofs: the DAG depends on the validator set, which depends on the blockchain, which depends on the DAG. This makes it harder to assess correctness. It is not obvious if and how the correctness proofs for fixed validator sets, and the associated bounds on faulty validators, generalize to these dynamic validator sets, and whether there should also be limitations on the changes to validator sets within a single block. Another complication is how bounds on faulty validators generalize from validator count to validator stake, since the unit of correctness or faultiness is still a whole validator, not a unit of stake. Some empirical evidence for these complexities is that an early version of AleoBFT suffered from a subtle flaw that would allow the blockchain to fork, as reported in [45, finding #02], further motivating a machine-checkable proof.

The work described in this paper addresses the issues raised above. We developed a formal model of a DAG-based BFT consensus protocol with dynamic stake, based on AleoBFT but arguably applicable to similar protocols. We formally proved, in the ACL2 theorem prover [22], that blockchains never fork in this model, under the assumption that each validator set that arises during execution satisfies the less-than-one-third bound on faulty validator stake. No other limitation is needed: validator sets can change completely at every block, and the use of validator stake in the protocol's decisions is independent from validator count. We modeled the protocol as a labeled state transition system. We defined and proved a number of state invariants that lead to blockchain nonforking. The proofs of the invariants are by induction: they apply to arbitrarily long executions and arbitrarily large system states; they are verified efficiently by ACL2 (in 1 minute), without costly search space explorations. Our ACL2 model and proofs are available open-source [14]; they consist of about 20,000 lines, including extensive documentation.

To our knowledge, ours is the first machine-checked proof of blockchain nonforking of a DAG-based BFT consensus protocol with dynamic stake, that applies to arbitrarily long executions and arbitrarily large system states.

Our formally modeled and verified protocol is based on Narwhal [19] for DAG construction and Bullshark [39, 40] for blockchain construction. Each validator constructs its DAG and blockchain through successive rounds. The circular dependency between DAG and blockchain construction is handled by the protocol via a *lookback* approach: the validator set in charge of each round is determined only by blocks generated up to ℓ rounds earlier, where ℓ is a fixed lookback distance. When validators bond or unbond in a block at a certain round, they are not yet or still in charge until ℓ rounds later: there is a delay, measured in ℓ rounds, between a bonding or unbonding transaction and the actual joining or leaving of the validator set. In the formal proofs, the seeming circularity between DAG and blockchain correctness is resolved by an induction proof on both invariants at the same time.

Section 2 presents our formal model of the protocol, and Section 3 describes the key aspects of the proofs; formal definitions of all the invariants, and sketches of their proofs, are in Appendix B. These sections and appendix use a generic mathematical notation, summarized in Appendix A, that should be more widely accessible than ACL2. Section 4 briefly overviews how model and proofs are formalized in ACL2, and Appendix D shows samples of the ACL2 formalization; some background on ACL2 is given in Appendix C. After discussing related work in Section 5, some closing remarks are given in Section 6.

2 Formal Model

The protocol is modeled as a labeled state transition system, which moves from state to state via nondeterministic events. The system state represents the states of the individual validators and of the network that connects them. When an event happens, it affects certain parts of the system state. A labeled state transition system formulation is more directly usable in a theorem prover than pseudocode, which would need a formalized semantics.

This labeled state transition system models the protocol at an abstract level that is appropriate to the purpose of proving blockchain nonforking. The model has a larger set of possible executions (i.e. sequences of states linked by events) than an actual implementation. But properties proved to hold for every execution of the model also apply to every subset of such executions.[2]

[2]To prove other kinds of properties (e.g. liveness), the model would need to be augmented to capture additional features of the implementation, restricting the possible executions.

Addresses:	$a \in A$
Round numbers:	$r \in R = \{1, 2, 3, \ldots\}$
Round numbers and 0:	$r^0 \in R^0 = R \cup \{0\}$
Stake amounts:	$k \in K = \{1, 2, 3, \ldots\}$
Stake amounts and 0:	$k^0 \in K^0 = K \cup \{0\}$
Bonding transactions:	$x = \langle \textsc{bond}, a, k \rangle \in X_{\text{B}} = \{\textsc{bond}\} \times A \times K$
Unbonding transactions:	$x = \langle \textsc{unbond}, a \rangle \in X_{\text{U}} = \{\textsc{unbond}\} \times A$
Other transactions:	$x \in X_{\text{O}}$
All transactions:	$x \in X = X_{\text{B}} \cup X_{\text{U}} \cup X_{\text{O}}$
Blocks:	$b = \langle r, \overline{x} \rangle \in B = R \times \overline{X}$
Certificates:	$c = \langle a, r, \overline{x}, \widetilde{p}, \widetilde{q} \rangle \in C = A \times R \times \overline{X} \times \widetilde{A} \times \widetilde{A}$
Endorsed pairs:	$d = \langle a, r \rangle \in D = A \times R$
DAGs:	$g \in G = \widetilde{C}$
Validator states:	$v = \langle r, g, \widetilde{d}, l, \overline{b}, \widetilde{c} \rangle \in V = R \times G \times \widetilde{D} \times R^0 \times \overline{B} \times \widetilde{C}$
Messages:	$m = \langle c, a \rangle \in M = C \times A$
System states:	$s = \langle \vec{v}, \widetilde{m} \rangle \in S = (A \rightsquigarrow V) \times \widetilde{M}$
Committees:	$w \in W = A \rightsquigarrow K$
Certificate creation events:	$e = \langle \textsc{create}, c \rangle \in E_{\text{CC}} = \{\textsc{create}\} \times C$
Certificate acceptance events:	$e = \langle \textsc{accept}, m \rangle \in E_{\text{CA}} = \{\textsc{accept}\} \times M$
Round advancement events:	$e = \langle \textsc{advance}, a \rangle \in E_{\text{RA}} = \{\textsc{advance}\} \times A$
Anchor commitment events:	$e = \langle \textsc{commit}, a \rangle \in E_{\text{AC}} = \{\textsc{commit}\} \times A$
All events:	$e \in E = E_{\text{CC}} \cup E_{\text{CA}} \cup E_{\text{RA}} \cup E_{\text{AC}}$
Initial states:	$I = \{\langle \vec{v}, \varnothing \rangle \in S \mid \forall a \in \mathcal{D}(\vec{v}) : \vec{v}(a) = \langle 1, \varnothing, \varnothing, 0, \epsilon, \varnothing \rangle\}$

Figure 1: States and Events

2.1 Labeled State Transition System

The *labeled state transition system* is the tuple $\langle S, E, I, T \rangle$, where S is the set of *states*, E is the set of *events*, $I \subseteq S$ is the set of *initial states*, and $T \subseteq S \times E \times S$ is the *transition relation* among old states, events, and new states. $T(s, e, s')$ holds exactly when the event e causes a transition from the old state s to the new state s'. The *execution relation* $T^* \subseteq S \times \overline{(E \times S)}$, derived from T, holds on $\langle s_0, [\langle e_1, s_1 \rangle, \ldots, \langle e_n, s_n \rangle] \rangle$, where $n \geq 0$, exactly when $T(s_{i-1}, e_i, s_i)$ for every i such that $1 \leq i \leq n$: it describes how a sequence of zero or more events $[e_1, \ldots, e_n] \in \overline{E}$ moves the system through a sequence of one or more states $[s_0, \ldots, s_n] \in \overline{S}$. The set of states *reachable from* a state $s_0 \in S$ is $H(s_0) = \{s_n \in S \mid \exists\, e_1, \ldots, e_n, s_1, \ldots, s_{n-1} : T^*(s_0, [\langle e_1, s_1 \rangle, \ldots, \langle e_n, s_n \rangle])\}$; note that $s_0 \in H(s_0)$. The set of *reachable* states consists of the ones reachable from initial states, i.e. $H = \{s \in H(s_0) \mid s_0 \in I\}$; note that $I \subseteq H$.

Figure 1 defines the S, E, and I components of the labeled state transition system, along with their constituents; while committees are not explicitly part of the states, they are derived from states, as described later. The T component of the labeled state transition system is inductively defined in Figure 2 as the smallest relation satisfying the inference rules in the figure: for each rule, for every satisfying assignment of the variables in the premises above the line, the conclusion below the line adds an element to the transition relation. The inference rules make use of auxiliary constants, functions, and relations, which are introduced in Figure 3. The rest of this section explains the formal definitions in the figures and how they relate to an implementation.

2.2 Validators

The protocol is run by *validators*, each of which has a unique and immutable *address* $a \in A$. This address is essentially a public key (but the details are irrelevant to our model, so Figure 1 does not define A), whose associated private key is used by the validator to sign data as described later.

A validator is *correct* if it always follows the protocol and its private key is not compromised; otherwise it is *faulty*. The (Byzantine) abilities of faulty validators are characterized later.

Each correct validator has an internal *validator state* $v \in V$ whose components (described later) contain information needed to participate to the protocol correctly. A system state $s \in S$ includes a finite map $\vec{v} \in A \rightsquigarrow V$, each of whose entries represents the address and state of a correct validator. Faulty validators

Certificate creation (by correct validator)

$$\frac{\begin{array}{c} s = \langle \vec{v}, \widetilde{m} \rangle \qquad c = \langle a, r, \overline{x}, \widetilde{p}, \widetilde{q} \rangle \qquad a \in \mathcal{D}(\vec{v}) \qquad v = \vec{v}(a) \qquad v.r = r \qquad r = 1 \iff \widetilde{p} = \varnothing \\ \nexists\, c' \in v.g : \langle c'.a, c'.r \rangle = \langle a, r \rangle \\ r \neq 1 \implies \mathit{isClosed}(\widetilde{p}, r-1, v.g) \wedge \mathit{isQuorum}(\widetilde{p}, r-1, v.\overline{b}) \\ a \notin \widetilde{q} \qquad \mathit{isQuorum}(\{a\} \cup \widetilde{q}, r, v.\overline{b}) \\ \forall q \in \widetilde{q} \cap \mathcal{D}(\vec{v}) : \mathit{isNew}(a, r, \vec{v}(q)) \\ r \neq 1 \implies \forall q \in \widetilde{q} \cap \mathcal{D}(\vec{v}) : (\mathit{isClosed}(\widetilde{p}, r-1, \vec{v}(q).g) \wedge \mathit{isQuorum}(\widetilde{p}, r-1, \vec{v}(q).\overline{b})) \\ v' = v\langle g \mapsto v.g \cup \{c\} \rangle \qquad \vec{v}' = \vec{v}\{a \mapsto v'\} \qquad \vec{v}'' = \mathit{endorse}(\vec{v}', \widetilde{q}, \langle a, r \rangle) \\ \widetilde{m}' = \widetilde{m} \cup \{\langle c, a' \rangle \mid a' \in \mathcal{D}(\vec{v}) \setminus \{a\}\} \qquad s' = \langle \vec{v}'', \widetilde{m}' \rangle \end{array}}{\langle s, \langle \textsc{create}, c \rangle, s' \rangle \in T}$$

Certificate creation (by faulty validator)

$$\frac{\begin{array}{c} s = \langle \vec{v}, \widetilde{m} \rangle \qquad c = \langle a, r, \overline{x}, \widetilde{p}, \widetilde{q} \rangle \qquad a \notin \mathcal{D}(\vec{v}) \qquad \widetilde{q} \cap \mathcal{D}(\vec{v}) \neq \varnothing \implies (r = 1 \iff \widetilde{p} = \varnothing) \\ \forall q \in \widetilde{q} \cap \mathcal{D}(\vec{v}) : \mathit{isNew}(a, r, \vec{v}(q)) \\ r \neq 1 \implies \forall q \in \widetilde{q} \cap \mathcal{D}(\vec{v}) : (\mathit{isClosed}(\widetilde{p}, r-1, \vec{v}(q).g) \wedge \mathit{isQuorum}(\widetilde{p}, r-1, \vec{v}(q).\overline{b})) \\ \vec{v}' = \mathit{endorse}(\vec{v}, \widetilde{q}, \langle a, r \rangle) \qquad \widetilde{m}' = \widetilde{m} \cup \{\langle c, a' \rangle \mid a' \in \mathcal{D}(\vec{v})\} \qquad s' = \langle \vec{v}', \widetilde{m}' \rangle \end{array}}{\langle s, \langle \textsc{create}, c \rangle, s' \rangle \in T}$$

Certificate acceptance

$$\frac{\begin{array}{c} s = \langle \vec{v}, \widetilde{m} \rangle \qquad m = \langle c, a \rangle \in \widetilde{m} \qquad a \in \mathcal{D}(\vec{v}) \qquad v = \vec{v}(a) \\ c.r \neq 1 \implies \mathit{isClosed}(c.\widetilde{p}, c.r-1, v.g) \\ c.a \notin c.\widetilde{q} \qquad \mathit{isQuorum}(\{c.a\} \cup c.\widetilde{q}, c.r, v.\overline{b}) \\ v' = v\langle g \mapsto v.g \cup \{c\} \rangle \langle \widetilde{d} \mapsto v.\widetilde{d} \setminus \{\langle c.a, c.r \rangle\} \rangle \qquad \vec{v}' = \vec{v}\{a \mapsto v'\} \qquad \widetilde{m}' = \widetilde{m} \setminus \{m\} \qquad s' = \langle \vec{v}', \widetilde{m}' \rangle \end{array}}{\langle s, \langle \textsc{accept}, m \rangle, s' \rangle \in T}$$

Round advancement

$$\frac{\begin{array}{c} s = \langle \vec{v}, \widetilde{m} \rangle \qquad a \in \mathcal{D}(\vec{v}) \qquad v = \vec{v}(a) \\ v' = v\langle r \mapsto v.r + 1 \rangle \qquad \vec{v}' = \vec{v}\{a \mapsto v'\} \qquad s' = \langle \vec{v}', \widetilde{m} \rangle \end{array}}{\langle s, \langle \textsc{advance}, a \rangle, s' \rangle \in T}$$

Anchor commitment

$$\frac{\begin{array}{c} s = \langle \vec{v}, \widetilde{m} \rangle \qquad a \in \mathcal{D}(\vec{v}) \qquad v = \vec{v}(a) \qquad v.r \bmod 2 = 1 \qquad v.r \neq 1 \qquad v.l < v.r - 1 \\ \mathit{isAnchor}(c, v.g, v.\overline{b}) \qquad c.r = v.r - 1 \qquad \mathit{isElected}(c, v.g, v.\overline{b}) \\ \overline{c} = \mathit{collAnch}(c, v.l, v.g, v.\overline{b}) \qquad \langle \overline{b}', \widetilde{c}' \rangle = \mathit{genBlk}(\overline{c}, v.g, v.\overline{b}, v.\widetilde{c}) \\ v' = v\langle l \mapsto c.r \rangle \langle \overline{b} \mapsto \overline{b}' \rangle \langle \widetilde{c} \mapsto \widetilde{c}' \rangle \qquad \vec{v}' = \vec{v}\{a \mapsto v'\} \qquad s' = \langle \vec{v}', \widetilde{m} \rangle \end{array}}{\langle s, \langle \textsc{commit}, a \rangle, s' \rangle \in T}$$

Figure 2: Transitions

Last block round: $last : \overline{B} \to R^0 \qquad last(\epsilon) = 0 \qquad last(\overline{b} \bowtie [b]) = b.r$

Committee change: $cmt : W \times (X \cup \overline{X} \cup B \cup \overline{B}) \to W$

$$\begin{cases} cmt(w, \langle \text{BOND}, a, k\rangle) = w\{a \mapsto k\} & \Longleftarrow a \notin \mathcal{D}(w) \\ cmt(w, \langle \text{BOND}, a, k\rangle) = w\{a \mapsto k + w(a)\} & \Longleftarrow a \in \mathcal{D}(w) \\ cmt(w, \langle \text{UNBOND}, a\rangle) = w{\downarrow_{\mathcal{D}(w)\setminus\{a\}}} & \\ cmt(w, x) = w & \Longleftarrow x \in X_{\text{O}} \\ cmt(w, b) = cmt(w, b.\overline{x}) & \\ cmt(w, \epsilon) = w & \\ cmt(w, [z] \bowtie \overline{z}) = cmt(cmt(w, z), \overline{z}) & \end{cases}$$

Genesis committee: $gcmt \in W$

Bonded committee: $bcmt : R \times \overline{B} \to W \cup \{\bot\}$

$$\begin{cases} bcmt(r, \overline{b}) = \bot & \Longleftarrow r > last(\overline{b}) + 2 \\ bcmt(r, \overline{b}) = bcmt'(r, \overline{b}) & \Longleftarrow r \leq last(\overline{b}) + 2 \end{cases}$$

$bcmt' : R \times \overline{B} \to W$

$$\begin{cases} bcmt'(r, \epsilon) = gcmt & \\ bcmt'(r, \overline{b}) = cmt(gcmt, \overline{b}) & \Longleftarrow r > last(\overline{b}) \\ bcmt'(r, \overline{b} \bowtie [b]) = bcmt'(r, \overline{b}) & \Longleftarrow r \leq b.r \end{cases}$$

Lookback amount: $lkbk \in \{1, 2, 3, \ldots\}$

Active committee: $acmt : R \times \overline{B} \to W \cup \{\bot\}$

$$\begin{cases} acmt(r, \overline{b}) = bcmt(r - lkbk, \overline{b}) & \Longleftarrow r > lkbk \\ acmt(r, \overline{b}) = gcmt & \Longleftarrow r \leq lkbk \end{cases}$$

Total stake: $nstk : W \to K^0 \qquad nstk(w) = \sum_{a \in \mathcal{D}(w)} w(a)$

Max faulty stake: $fstk : W \to K^0$

$$\begin{cases} fstk(w) = \max\{f \in K^0 \mid f < nstk(w)/3\} & \Longleftarrow nstk(w) \neq 0 \\ fstk(w) = 0 & \Longleftarrow nstk(w) = 0 \end{cases}$$

Quorum stake: $qstk : W \to K^0 \qquad qstk(w) = nstk(w) - fstk(w)$

DAG edge: $isEdge \subseteq C \times C \times G \qquad isEdge(c, c', g) \Longleftrightarrow c \in g \wedge c' \in g \wedge c.r = c'.r + 1 \wedge c'.a \in c.\widetilde{p}$

DAG path: $isPath \subseteq C \times C \times G$

$isPath(c, c', g) \Longleftrightarrow \exists\, c_1, \ldots, c_n : (n > 0 \wedge c_1 = c \wedge c_n = c' \wedge \forall i \in \{2, \ldots, n\} : isEdge(c_i, c_{i-1}, g))$

Closure check: $isClosed \subseteq \widetilde{A} \times R \times G \qquad isClosed(\widetilde{a}, r, g) \Longleftrightarrow \forall a \in \widetilde{a} : \exists\, c \in g : \langle c.a, c.r\rangle = \langle a, r\rangle$

Quorum check: $isQuorum \subseteq \widetilde{A} \times R \times \overline{B}$

$isQuorum(\widetilde{a}, r, \overline{b}) \Longleftrightarrow \exists\, w : (w = acmt(r, \overline{b}) \neq \bot \wedge \widetilde{a} \subseteq \mathcal{D}(w) \wedge \sum_{a \in \widetilde{a}} w(a) \geq qstk(w))$

Newness check: $isNew \subseteq A \times R \times V \qquad isNew(a, r, v) \Longleftrightarrow (\nexists\, c \in v.g : \langle c.a, c.r\rangle = \langle a, r\rangle) \wedge \langle a, r\rangle \notin v.\widetilde{d}$

Endorsement: $endorse : (A \rightsquigarrow V) \times \widetilde{A} \times D \to (A \rightsquigarrow V)$

$$\begin{cases} endorse(\vec{v}, \varnothing, d) = \vec{v} & \\ endorse(\vec{v}, \{q\} \cup \widetilde{q}, d) = endorse(\vec{v}, \widetilde{q}, d) & \Longleftarrow q \notin \mathcal{D}(\vec{v}) \\ endorse(\vec{v}, \{q\} \cup \widetilde{q}, d) = endorse(\vec{v}\{q \mapsto v\langle \widetilde{d} \mapsto v.\widetilde{d} \cup \{d\}\rangle\}, \widetilde{q}, d) & \Longleftarrow q \in \mathcal{D}(\vec{v}) \wedge v = \vec{v}(q) \end{cases}$$

Leader selection: $leader : W \times R \to A \qquad \mathcal{D}(w) \neq \varnothing \Longrightarrow leader(w, r) \in \mathcal{D}(w)$

Anchor: $isAnchor \subseteq C \times G \times \overline{B}$

$isAnchor(c, g, \overline{b}) \Longleftrightarrow c \in g \wedge \exists\, w : (w = acmt(c.r, \overline{b}) \neq \bot \wedge \mathcal{D}(w) \neq \varnothing \wedge c.a = leader(\mathcal{D}(w), c.r))$

Election check: $isElected \subseteq C \times G \times \overline{B}$

$isElected(c, g, \overline{b}) \Longleftrightarrow \exists\, w, \widetilde{a} : (w = acmt(c.r + 1, \overline{b}) \neq \bot \wedge \widetilde{a} = \{c'.a \mid isEdge(c', c, g)\} \wedge$
$\widetilde{a} \subseteq \mathcal{D}(w) \wedge \sum_{a \in \widetilde{a}} w(a) > fstk(w))$

Previous anchor: $isPrevAnch \subseteq C \times C \times G \times \overline{B}$

$isPrevAnch(c, c', g, \overline{b}) \Longleftrightarrow isAnchor(c, g, \overline{b}) \wedge isAnchor(c', g, \overline{b}) \wedge isPath(c, c', g) \wedge c'.r < c.r \wedge$
$\nexists\, c'' : (isAnchor(c'', g, \overline{b}) \wedge isPath(c, c'', g) \wedge c'.r < c''.r < c.r)$

Anchor collection: $collAnch : C \times R^0 \times G \times \overline{B} \to \overline{C}$

$$\begin{cases} collAnch(c, l, g, \overline{b}) = [c] & \Longleftarrow \nexists\, c' : (isPrevAnch(c, c', g, \overline{b}) \wedge l < c'.r) \\ collAnch(c, l, g, \overline{b}) = collAnch(c', l, g, \overline{b}) \bowtie [c] & \Longleftarrow isPrevAnch(c, c', g, \overline{b}) \wedge l < c'.r \end{cases}$$

Certificate ordering: $orderCert : \widetilde{C} \to \overline{C} \qquad orderCert(\widetilde{c}) = [c_1, \ldots, c_n] \Longrightarrow \widetilde{c} = \{c_1, \ldots, c_n\} \wedge \forall i, j : (1 \leq i < j \leq n \Longrightarrow c_i \neq c_j)$

Transaction collection: $collTrans : \overline{C} \to \overline{X}$

$$\begin{cases} collTrans(\epsilon) = \epsilon & \\ collTrans([c] \bowtie \overline{c}) = c.\overline{x} \bowtie collTrans(\overline{c}) & \end{cases}$$

Block generation: $genBlk : \overline{C} \times G \times \overline{B} \times \widetilde{C} \to \overline{B} \times \widetilde{C}$

$$\begin{cases} genBlk(\epsilon, g, \overline{b}, \widetilde{c}) = \langle \overline{b}, \widetilde{c}\rangle & \\ genBlk([c] \bowtie \overline{c}, g, \overline{b}, \widetilde{c}) = genBlk(\overline{c}, g, \overline{b}', \widetilde{c}') & \Longleftarrow \widetilde{c}' = \{c' \mid isPath(c, c', g)\} \wedge \\ & \qquad \overline{b}' = \overline{b} \bowtie [\langle c.r, collTrans(orderCert(\widetilde{c}' \setminus \widetilde{c}))\rangle] \end{cases}$$

Figure 3: Auxiliary Constants, Functions, and Relations

are not an explicit part of system states: their behavior is represented, as described later, in terms of the possible effects that they may have on correct validators, which are those that must reach consensus.[3]

At any point of the execution of the protocol, a committee of validators is in charge (as detailed later); the committee is *dynamic*, potentially changing at every block, and may include both correct and faulty validators. The domain $\mathcal{D}(\vec{v})$ includes all the correct validators that may be part of any committee at any time; they are not just the ones in the active committee. The active committee is not a global notion, but is local to each validator (as described later). $\mathcal{D}(\vec{v})$ never changes in our model; as defined in I in Figure 1, there is one initial state for each possible choice of $\mathcal{D}(\vec{v})$, and the rules in Figure 2 are such that every reachable state has the same $\mathcal{D}(\vec{v})$.

In an implementation, validators actually come into and out of existence (from the point of view of the protocol), which would be more naturally modeled by letting $\mathcal{D}(\vec{v})$ change. But new validators must sync their internal state with existing validators before they actively participate to the protocol. Our model captures this syncing process indirectly, by pretending that all future validators are already in the system, keeping their internal states in sync with the ones in the current and past committees, but actively participating only when they are in the committee.

2.3 Network

Validators communicate via a *network* that provides authenticated point-to-point connections with unbounded delays. This is in line with common network assumptions in the BFT literature.

A *message* $m \in M$ consists of a certificate (described later) and the address of the validator to which the message is destined; the address of the sender is part of the certificate. A system state $s \in S$ includes a set of messages $\widetilde{m}$ that models the state of the network, consisting of the messages that have been sent but not yet received; initially this is the empty set, as defined in I in Figure 1.

As described later, some events add messages to the network, modeling their sending, while other events remove messages from the network, modeling their receiving. Since events are nondeterministic, after a message is added to the network (i.e. sent), it may be removed (i.e. received) arbitrarily later, if at all. A faulty validator cannot forge or modify a message from a correct validator, because that would require compromising the private key (see Section 2.2).

Each message is addressed to exactly one validator. The *broadcasting* of a message is modeled as the sending of multiple messages.

2.4 Rounds and Round Advancement

The protocol proceeds in *rounds*, numbered starting from 1; the use and significance of rounds is described later. A validator state $v \in V$ includes the number $r \in R$ of the round that the validator is at. This is initially 1, as defined in I in Figure 1.

In an implementation, the round number of a validator gets incremented (never decremented) under conditions that our model does not need to capture in detail, because the exact logic of round advancement does not affect blockchain nonforking and related properties. The fourth rule in Figure 2 says that an event $\langle \text{ADVANCE}, a \rangle$ increments the round of a validator with address a by one: given a system state $s = \langle \vec{v}, \widetilde{m} \rangle$, and an event $\langle \text{ADVANCE}, a \rangle$ that references the address a of a correct validator (i.e. such that $a \in \mathcal{D}(\vec{v})$), whose state is $v = \vec{v}(a)$, the new validator state v' is obtained by incrementing $v.r$ by one, the new validator map $\vec{v}'$ is obtained by updating the state of a to be v', and the new system state s is obtained by updating its validator map to be $\vec{v}'$; the network state $\widetilde{m}$ is unchanged. This rule captures a superset of all the possible round advancements in an implementation, which take place under more stringent conditions.

2.5 Blockchains

A validator state $v \in V$ includes the validator's own view of the *blockchain*, as a sequence $\overline{b} \in \overline{B}$ of blocks. Each *block* $b = \langle r, \overline{x} \rangle$ consists of the round at which the block is generated (as described later) and a sequence of *transactions*.

[3] Earlier versions of our formal model used a map $\vec{v} \in A \rightsquigarrow V \cup \{\bot\}$, where $\bot$ indicated a faulty validator. But removing that from the map made things simpler.

Transactions are the "centerpiece" of the protocol, whose purpose is to order transactions into the blockchain. Our formal model treats transactions as mostly opaque, with the exception of *bonding* and *unbonding* transactions, which, as described later, let validators join and leave committees. The set X of all transactions is partitioned into three (disjoint) sets X_{B}, X_{U}, and X_{O}.

The blocks in a blockchain $[b_1, \ldots, b_n]$ are ordered from left to right: b_1 is the oldest block, and b_n is the newest block. Blockchains normally start with a genesis block, but our model leaves that implicit: b_1 is the first block after the genesis block, and initially the blockchains of all validators are empty, as defined in I in Figure 1. As described later, blocks are generated only at (some) even rounds, at most one block per round; thus, the round numbers in a blockchain are all even, and strictly increase from left to right. The function *last* in Figure 3 returns the round of the last block in a blockchain, or 0 if the blockchain is empty.

2.6 Committees

A *committee* $w \in W$ is a finite map from the addresses of the validators that form the committee to the stakes associated to such validators: when a validator joins a committee, it does so with a certain stake; different validators in the same committee may have different stakes. As described later, stakes act like weights for the validators when making certain decisions in the protocol. The case of a committee whose validators all have equal stake is similar to more typical BFT systems where decisions are based on validator count instead of stake.

Committees change according to the bonding and unbonding transactions in a blockchain, as defined by the function *cmt* in Figure 3. A bonding transaction $\langle \text{BOND}, a, k \rangle$ adds a to the committee with stake k if a is not already in the committee; if a is already in the committe, k is added to the stake already present. An unbonding transaction $\langle \text{UNBOND}, a \rangle$ removes a from the committee if present; it is a no-op if a is not in the committee. Any other kind of transaction does not change the committee. A sequence of transactions changes the committee as expected, one transaction after the other. A block changes the committee via its transactions. A sequence of blocks changes the committee as expected, one block after the other.[4]

Since each validator has its own view of the blockchain, it also has its own view of committees. There is no global notion of "current committee" in the system, like there is no global notion of "current blockchain". However, as described in Section 3, a consequence of blockchain agreement is committee agreement among validators.

Validators start with a common *genesis committee* $gcmt \in W$ (whose specifics are irrelevant to our model, so Figure 3 does not define *gcmt*). Each validator has a *bonded committee* at each round r: it is the committee resulting from the bonding and unbonding transactions in the blocks whose rounds are before r. The bonded committee at rounds 1 and 2 is always the genesis committee, because no blocks have rounds before 2 (blocks always have even rounds). If there is a block at round 2, its bonding and unbonding transactions determine the bonded commitee at rounds 3 and 4, which otherwise is still the genesis committee. And so on: each block determines, together with the blocks that precede it, the bonded committee at the two rounds just after it, and possibly at later rounds, until there is either another block or no more blocks.

The bonded committee at round r, given a blockchain $\overline{b}$, is defined by the function *bcmt* in Figure 3. The blockchain $\overline{b}$ determines the bonded committees only up to round $last(\overline{b}) + 2$: the function *bcmt* returns $\perp$ for larger rounds, meaning that a validator cannot (yet) calculate later bonded committees, because a new block may be added at round $last(\overline{b}) + 2$. The auxiliary function $bcmt'$ is used by *bcmt* to calculate the bonded committee at all the applicable rounds, by recursively finding the blockchain prefix that applies to the round and using *cmt* on it.

As described later, a validator needs to make decisions for a round based on the committee in charge of the round, but sometimes the validator cannot yet calculate the bonded committee. For this reason, the protocol uses a *lookback* approach: the committee in charge of a round is the one bonded at a fixed-distance earlier round. We model the distance as the positive integer *lkbk* in Figure 3, whose exact definition is irrelevant to our model. This leads to the notion of *active committee* at a round, i.e. the committee in charge of the round, defined by the function *acmt* in Figure 3; the genesis committee is the active one for all the rounds up to at least $lkbk + 2$.

[4] Bonding an already bonded validator, or unbonding a non-bonded validator, could be regarded as invalid transactions. But we keep our model simpler by regarding all transactions as valid, as this does not affect the salient aspects of our formal proofs.

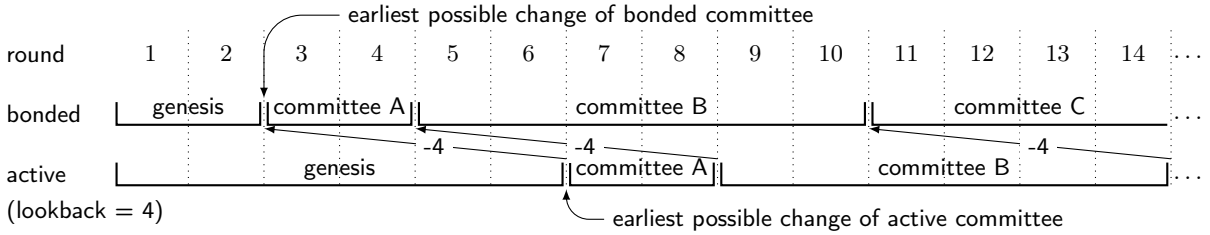

Figure 4: Example of Bonded and Active Committees

Figure 4 depicts an example of bonded and active committees. The bonded committee at rounds 1 and 2 is the genesis committe, as always. The earliest possible change to the bonded committee may happen between rounds 2 and 3, via transactions in a block at round 2. This is the case in the example in the figure: committee A is the bonded one at rounds 3 and 4. The example also shows a bonded committee change between rounds 4 and 5, via transactions in the block at round 4. Committee B persists until round 10, because blocks at rounds 6 and 8 are absent or do not contain bonding or unbonding transactions. The block at round 10 makes committee C the bonded committee at round 11, and so on. Since the example assumes $lkbk = 4$, active committees are shifted by exactly 4 rounds, with the genesis committee filling the first 4 rounds. In an implementation, *lkbk* has a much larger value; the small value 4 has been chosen to ease illustration.

When a validator bonds (at a block's round), it waits *lkbk* rounds before actively participating to the protocol. When a validator unbonds (at a block's round), it waits *lkbk* rounds before stopping its active participation. This is the case for correct validators; faulty validators may attempt to participate at any time, but correct validators detect and curb such attempts, as described later.

The *total stake* of a committee is the sum of the stakes of all its members, as defined by the function *nstk* in Figure 3. This generalizes the total count of validators in typical BFT systems, often denoted as n; when each committee member has one unit of stake, our general notion specializes to that one.

The *maximum faulty stake* of a committee is the maximum sum of the stakes of the faulty members in the committee that the protocol can tolerate and still ensure consensus among correct validators (as our proofs show); it is defined by the function *fstk* in Figure 3. This generalizes the maximum count of faulty validators in typical BFT systems, often denoted as f. Note that *fstk* is *not* the sum of the stakes of the actual faulty members of the committee, which is not known to validators, who nonetheless need to use *fstk*, along with *nstk*, to run the protocol. In typical BFT systems, f is defined as the maximum integer strictly below $n/3$ (i.e. "less than 1/3 of the validators are faulty"); correspondingly, *fstk* is defined as the largest integer strictly less than 1/3 of *nstk*, defining it to be 0 when the committee is empty.[5] For non-empty committees, the definition of *fstk* in Figure 3 is equivalent to $fstk(w) = \lceil nstk(w)/3 \rceil - 1$ and to $fstk(w) = \lfloor (nstk(w) - 1)/3 \rfloor$.

The *quorum stake* of a committee is the difference between *nstk* and *fstk*, as defined by the function *qstk* in Figure 3. As described later, this is a threshold for certain decisions in the protocol, and is used to prove certain critical properties by quorum intersection. It corresponds to $n - f$ in typical BFT systems. The latter is equivalent to $2f + 1$ only assuming $n = 3f + 1$; but if instead $n = 3f + 2$ or $n = 3f + 3$, then $2f + 1$ is insufficient for quorum intersection. Restricting n to be $3f + 1$ is unrealistic, especially with dynamic committees, and unnecessary, since $n - f$ works in all cases.[6]

2.7 Certificates and DAGs

Before going into blocks, transactions are exchanged by validators via *certificates*. Validators create certificates and broadcast them, in messages, to other validators. A validator state $v \in V$ includes a set $g \in G$ of certificates created by that validator or created by and received from other validators; this set has the structure of a DAG, as described shortly.

[5] Our model allows empty committees, which may result from all the members unbonding in the same block. Although we have not formally studied the situation yet, it seems likely that the protocol would deadlock, with nobody actually running it.

[6] For the specific case $n = 3f + 3$, it would suffice to require a quorum stake of only $2f + 2$ instead of $2f + 3 = n - f$, but it is simpler to use $n - f$ uniformly.

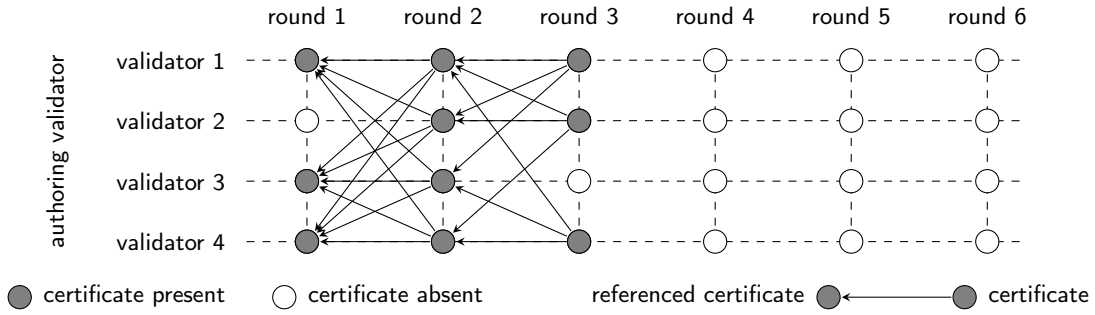

Figure 5: Example DAG of a Validator

A certificate $c \in C$ includes the address a of its *author*, i.e. the validator who created the certificate. In our model, this a represents a cryptographic signature with the validator's private key. Certificate authors can be thus relied upon: correct validators reject invalid signatures, and faulty validators cannot forge signatures of correct validators.

Correct validators author at most one certificate per round. The r component of a certificate $c \in C$ is the round in which c is created. Faulty validators may create multiple certificates for the same round, but the protocol ensures that correct validators accept at most (the same) one, rejecting the others: this is a crucial correctness property, as discussed in Section 3. Thus, the certificates in the DAG g of a validator have unique combinations of author and round, and can be arranged in a grid like Figure 5. Not all the positions in the grid need to have a certificate; positions are filled as the protocol runs, but some positions may never be filled.

Besides author a, round r, and a sequence of transactions $\overline{x}$, a certificate $c \in C$ includes a set $\widetilde{p}$ of references to *previous* DAG certificates. Each $p \in \widetilde{p}$ is the address of the author of a DAG certificate at the round just before r, which is uniquely determined, because of the aforementioned crucial property, by the combination of author p and round $r - 1$. As a special case, if $r = 1$, then $\widetilde{p} = \varnothing$: certificates at round 1 reference no previous certificates. These references to previous certificates are the *edges* of the DAG, depicted as backward-pointing arrows in Figure 5, and formalized by the predicate *isEdge* in Figure 3: there is an edge from c to c' in a DAG g exactly when both c and c' are in the DAG, c is one round after c', and the author of c' is referenced by c, i.e. $c'.a \in c.\widetilde{p}$. The *paths* in a DAG consists of zero or more edges, as formalized by the predicate *isPath* in Figure 3.

2.8 Certificate Creation

In an implementation of the protocol, folllowing Narwhal [19], certificate creation is a multi-step process: (i) a validator authors a *proposal*, which essentially consists of the first four components of a certificate in our model, and broadcasts it to the other validators; (ii) the other validators *endorse* the proposal (provided that it passes certain checks, as described later) by sending a signature for the proposal back the validator; (iii) when the validator collects enough endorsing signatures (details below), it creates and broadcasts a certificate. The $\widetilde{q}$ component of a certificate $c \in C$ consists of the addresses of the validators who endorsed the proposal. In our model, much like how the a component represents a signature by the author, each $q \in \widetilde{q}$ represents a signature by an *endorser*.

For simplicity, our model abstracts that multi-step process into one atomic event that creates and broadcasts a certificate, i.e. essentially the final act of the multi-step process. We have no separate notion of proposals, and no separate messages for sending proposals and endorsements. Our single-step model of certificate creation captures the critical aspects of the multi-step process.

Certificate creation is modeled by the first two rules in Figure 2, which distinguish between a correct author and a faulty author. These are fairly complex because they involve the multiple validators that take part in the multi-step process described above. In both rules, the event $\langle \textsc{create}, c \rangle$ contains the certificate c to create. In an implementation, the transactions in c are collected by validators from various sources (e.g. clients); the purpose of a proposal and resulting certificate is to include those collected transactions into the blockchain. We model this collection process abstractly, as the nondeterministic choice of the $\langle \textsc{create}, c \rangle$

event.

In the first rule, the author a is a correct validator, i.e. in $\mathcal{D}(\vec{v})$. Since a correct validator follows the protocol, it creates c only under the following conditions, expressed as premises in the rule:

- The round r of c matches the current round $v.r$.
- If $r = 1$, there are no references $\widetilde{p}$ to previous certificates, because there is no round 0; if $r \neq 1$, there is at least one reference. (Note the bidirectional implication.)
- The DAG $v.g$ has no certificate c' already authored by a for the same r, to ensure at most one certificate per round.
- If $r \neq 1$, the DAG $v.g$ contains, at round $r-1$, all the certificates authored by the elements in $\widetilde{p}$; this is expressed by the predicate *isClosed* in Figure 3. That is, c has no "dangling edges": its edges point to certificates in the DAG.
- If $r \neq 1$, the previous addresses in $\widetilde{p}$ form a quorum at round $r-1$, expressed by the predicate *isQuorum* in Figure 3: the validator a knows (i.e. can calculate via *acmt*) the active committee w at round $r-1$; every $p \in \widetilde{p}$ is in w; and the total stake of $\widetilde{p}$ in w is at least the quorum stake. The committee w is not empty; this is implied by $\widetilde{p} \neq \varnothing$, since $r \neq 1$.
- The author a is distinct from the endorsers in $\widetilde{q}$.
- The *signers* of c, i.e. the author a along with the endorsers in $\widetilde{q}$, form a quorum at round r, expressed using the same predicate *isQuorum* used above: the validator a knows the active committee at round r; the signers are members of the committee; and their total stake is at least the quorum stake. The signers' membership in the committee means that the author creates c only when in the active committee, and accepts endorsements only from members of the active committee.

An endorser $q \in \widetilde{q}$ may be correct or faulty. Since correct validators follow the protocol, a correct endorser, i.e. one in $\widetilde{q} \cap \mathcal{D}(\vec{v})$, endorses the certificate under the following conditions, also expressed as premises in the rule (if instead q is faulty, no conditions apply, because a faulty validator may sign anything):

- As expressed by the predicate *isNew* in Figure 3: the DAG $\vec{v}(q).g$ has no certificate with the same author and round as c; and the pair $\langle a, r \rangle$ of the author and round of c is not in the set $\vec{v}(q).\widetilde{d}$ of authors and pairs of certificates already endorsed by q. The purpose of this component of a validator state is to keep track of authors and rounds of endorsed certificates, to avoid endorsing different certificates with the same author and round; as described below, this set is updated with $\langle a, r \rangle$.
- If $r \neq 1$, the DAG $\vec{v}(q).g$ contains the certificates at the previous round referenced by $\widetilde{p}$. This is the same check performed by the author, via *isClosed*, but applied to the DAG of q.
- If $r \neq 1$, the previous addresses in $\widetilde{p}$ form a quorum at round $r-1$. This is the same check performed by the author, via *isQuorum*, but using the blockchain of q to calculate the committee.

The *isClosed* and *isQuorum* checks in the third and sixth lines of the rule are the same for author and endorser, but carried out on the respective internal states. The second line of the rule is the certificate newness check for the author, and corresponds to the fifth line, which covers the newness checks for the endorsers; the latter includes a check on endorsed pairs but the former does not, because validators only endorse certificates authored by others. The quorum check on the signers on the fourth line of the rule is only performed by the author: in the multi-step process described earlier, endorsers see only the proposal, while the author sees both the proposal and the endorsements, which it assembles into the certificate. As described later, validators (endorsers or not) perform the quorum check on signers when they receive a certificate from the network. The quorum check on previous certificate authors involves the committee at round $r-1$, because those certificates are at that round; the quorum check on the signers of c involves the committee at round r, because c is at that round.

If all the above conditions hold, the new system state s' is as follows, as defined by the remaining premises of the rule:

- The certificate c is added to the author's DAG $v.g$.
- The pair $\langle a, r \rangle$ is added to the set of endorsed pairs of every correct endorser, via the function *endorse* in Figure 3.
- The certificate c is broadcast to all the correct validators in the system except a (who already has c), by adding to the network a message with c for every such validator.

The rationale for sending messages to all correct validators, not just the ones in the committee (as seen by the author), is to model syncing, as described in Section 2.2.

In an implementation, author and endorsers also perform checks on the transactions $\overline{x}$ before proposing

or endorsing them. In our model all transactions are valid, for simplicity.

The second rule in Figure 2 models the creation of a certificate by a faulty validator, as expressed by the premise $a \notin \mathcal{D}(\vec{v})$. Faulty validators may generate any kind of proposal, but correct validators endorse it only if it passes the appropriate checks; in our model, certificates created by faulty validators omit checks by the author, but include checks from correct endorsers. The condition that c has no $p \in \widetilde{p}$ in round 1 but at least one in other rounds, holds only if there is at least one correct endorser, who checks that condition before endorsing; if all endorsers are faulty, that check is omitted altogether. Every correct endorser checks the same conditions as they do in the first rule in Figure 2, via *isNew*, *isClosed*, and *isQuorum*. There is no check that a is not in $\widetilde{q}$, and no *isQuorum* check on the signers: endorsers see only the proposal, not the endorsements; as described later, validators (endorsers or not) perform the quorum check on signers when they receive a certificate from the network.

If all the above conditions hold, the new system state s' is obtained by extending the sets of endorsed pairs of correct endorsers via *endorse*, and broadcasting the certificate to all the correct validators. The latter may seem unrealistic, since faulty validators may exclude some validators from the broadcast. However, our model does not require messages to be eventually delivered; they may sit in the network forever. Thus, changing the second rule in Figure 2 to broadcast the certificate to a subset of validators would make no difference to blockchain nonforking and related properties.[7] Furthermore, the currently modeled broadcast seems less unrealistic when considering that protocols like Narwhal [19] use a gossip approach to ensure, with high probability, that any certificate received by a correct validator is also received by all the other correct validators; if the certificate is not received by any correct validator, it has no impact on the consensus among correct validators, and it does not count as a CREATE event in our model.

2.9 Certificate Acceptance

Once certificates are broadcast as described in Section 2.8, receiving validators incorporate them into their DAGs under certain conditions, as formalized by the third rule in Figure 2. The message m in the network consists of a certificate c destined to a correct validator address a, whose internal state is v.

Unless the round of c is 1, the validator's DAG must already have all the certificates at the previous round referenced by c. This is checked via the same predicate *isClosed* used in the rules for certificate creation. For a validator who did not endorse c, without this check, adding c to the DAG could result in "dangling edges".

In addition, the validator ensures that the author of c is distinct from the endorsers of c, and that the total stake of author and endorsers (i.e. signers) is at least the quorum stake of the active committee at the round of c. This is checked via the same predicate *isQuorum* used in the rules for certificate creation.

The rule does not include a quorum check on $c.\widetilde{p}$. Under the fault tolerance conditions described in Section 3, that check would be redundant: the validator can rely on the correct signers having performed that check, and the quorum check on the signers ensures that at least one signer is correct. On the other hand, the quorum check on the signers is necessary: upon certificate creation, this is only performed by the author if correct, and not by the (correct) endorsers, for the reasons described in Section 2.8; thus, the validator accepting c cannot rely on any signer to have performed that check.

If all the above conditions hold, c is added to the DAG $v.g$ of the validator. In addition, the pair $\langle c.a, c.r \rangle$ is removed from the set of endorsed pairs $v.\widetilde{d}$. The pair is present in the set if the validator endorsed c: after adding c to the DAG, there is no longer a need to keep track of its author and pair. If the validator did not endorse c, the pair is already absent, and its removal is a no-op.

All correct validators can accept certificates, not only the validators in the active committee. This is in line with our model of syncing, discussed in Section 2.2.

2.10 Anchor Committment

Blocks are generated at (a subset of the) even rounds, at most one block per even round. The process is formalized by the fifth rule in Figure 2, which applies to a correct validator with address $a \in \mathcal{D}(\vec{v})$ and state $v = \vec{v}(a)$.

[7] It would make a difference for certain liveness properties.

A validator state $v \in V$ includes the number $l \in R^0$ of the *last committed round*, i.e. the round of the latest block generated by the validator, or 0 if no block has been generated yet. This is initially 0, as defined in I in Figure 1.

A block may be generated when the validator is at an odd round $v.r$ that is not 1; the round of the block is the even round just before it, i.e. $v.r - 1$. This even round must be strictly after the last committed round $v.l$, because otherwise a block has already been generated. These conditions are stated in the first line of the rule.

Each even round has a *leader* validator, chosen among the members of the active committee, via a random-like but deterministic selection, modeled by the function *leader* in Figure 3. The selection could be based on a hash of the committee and the round, but the details are irrelevant to our model; thus Figure 3 does not provide a definition, but only states that if the committee is not empty, *leader* returns a member of the committee.

The DAG $v.g$ of the validator may or may not have, at the even round $v.r - 1$, the certificate authored by the leader of that round. If it does, the certificate is called an *anchor*, as formalized by the predicate *isAnchor* in Figure 3: the predicate says that c is an anchor in DAG g with blockchain $\overline{b}$ exactly when c is in the DAG, there is a non-empty active committee at the round of the certificate, and the certificate's author is the leader at that round. The first two conditions in the second line of the fifth rule in Figure 2 say that there is an anchor c at the even round just before $v.r$.

The anchor c undergoes an *election*, whose *voters* are the certificates in the DAG at the odd round $v.r$ just after c: a voter whose $\widetilde{p}$ component includes the author of c counts as a 'yes' vote; a voter whose $\widetilde{p}$ component does not include the author of c counts as a 'no' vote. The election is won exactly when the total stake of the 'yes' votes is strictly more than the maximum faulty stake *fstk*. The election victory is formalized by the predicate *isElected* in Figure 3: w is the active committee at the odd round just after the certificate, and $\widetilde{a}$ is the set of validators, in the committee, whose authored certificates at that odd round count as 'yes' votes.

If the anchor c wins the election, it is *committed*, i.e. a block is generated from it, in the manner described shortly. But besides c, additional anchors at earlier rounds may be committed, as follows. If the DAG has an anchor c' at round $c.r - 2$ (i.e. c' is authored by the leader of round $c.r - 2$) that has not been already committed (i.e. $c.r - 2 > v.l$), and if there is a path from c to c', then c' is committed as well; if instead there is no such c', or no path from c to it, the round $c.r - 2$ is skipped, and a c' in the DAG with a path from c to it is sought at rounds $c.r - 4$, $c.r - 6$, etc., until either one is found, or $v.l$ is reached. If a suitable c' is found, the process continues recursively, with c' playing the role of c. The process always terminates, and results in a sequence of one or more anchors to commit, $[\ldots, c'', c', c]$, with c always the last one, possibly the only one. The process is formalized by the function *collAnch* in Figure 3, which makes use of the predicate *isPrevAnch* in Figure 3. The predicate *isPrevAnch* says that there is a DAG path from an anchor c to an anchor c' at a strictly earlier round, and that c' is the closest such anchor to c (i.e. no other anchor c'' can be found, with a path to it from c, at a round strictly between c and c'). The function *collAnch* recursively collects the sequence of anchors to commit, starting with c; the sequence has increasing round numbers from left to right. In the fifth rule in Figure 2, the sequence is $\overline{c}$.

Figure 6 shows an example of committed and skipped anchors, assuming 4 validators with equal stake. When the validator, whose DAG is shown in the figure, is at round 3, it can commit the anchor for round 2 as soon as the anchor has two 'yes' votes. When the validator is at round 5, the anchor for round 4 is not committed (yet), because it has only one 'yes' vote. When the validator is at round 7, the anchor for round 6 is not committed, because it is absent. When the validator is at round 9, the anchor for round 8 is not committed, because it has only one 'yes' vote. When the validator is at round 11, the anchor for round 10 is committed, because it has two 'yes' votes; since there is no path to the anchors for rounds 8 and 6, these two anchors are skipped permanently; since there is a path to the anchor for round 4, this anchor is committed as well.

One block is generated for each element in $\overline{c}$, from left to right, as formalized by the function *genBlk* in Figure 3, which makes use of the functions *orderCert* and *collTrans* in Figure 3. The function *orderCert* deterministically orders a set of certificates; the details of the order are irrelevant, so Figure 3 does not define this function, but only says that *orderCert*$(\widetilde{c})$ puts the certificates from $\widetilde{c}$ into a sequence without repetitions. The function *collTrans*$(\overline{c})$ concatenates together all the transactions from the sequence of certificates $\overline{c}$. The function *genBlk*$(\overline{c}, g, \overline{b}, \widetilde{c})$ extends $\overline{b}$ with one block for each certificate c in the sequence $\overline{c}$. The block consists

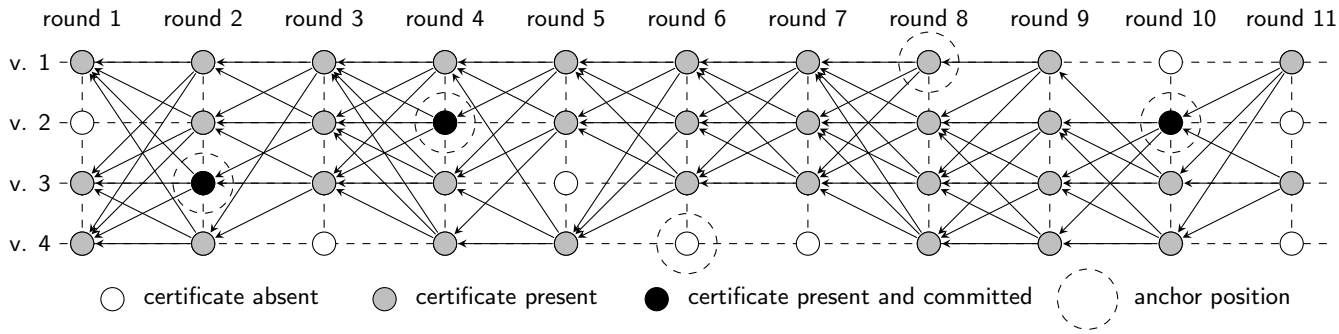

Figure 6: Example of Committed and Skipped Anchors

of the round of c and of all the transactions from the *causal history* of c, i.e. all the certificates reachable from c (including c itself) via paths in the DAG g, except for the certificates whose transactions have been already put into the blockchain; these already committed certificates are the ones in the set $\widetilde{c}$. In the definition of *genBlk* in Figure 3, $\widetilde{c}'$ is the causal history of c; the already committed certificates $\widetilde{c}$ are removed from it, the resulting set is ordered, and its transactions put into the block. Besides returning the updated blockchain, *genBlk* also returns the updated set of committed certificates.

A validator state $v \in V$ includes the set $\widetilde{c}$ of certificates committed so far to the blockchain; it is initially empty, as defined in I in Figure 1. In the fifth rule in Figure 2, the collected anchors $\overline{c}$ are passed to *genBlk*, along with the validator's DAG $v.g$, blockchain $v.\overline{b}$, and committed certificates $v.\widetilde{c}$, obtaining an updated blockchain $\overline{b}'$ and updated committed certificates $\widetilde{c}'$. These two are used to update the validator's state, along with updating $v.l$ to the round $c.r$ of the last committed anchor.

All correct validators can commit anchors, not only the validators in the active committee. This is in line with our model of syncing, discussed in Section 2.2.

2.11 Execution

The system starts in an initial state $s_0 \in I$, which is completely determined by the choice of addresses of correct validators $\mathcal{D}(\vec{v})$. Initially, each validator is in round 1, has an empty DAG, has endorsed no author-round pairs, has 0 as the last committed round, has an empty blockchain, and has no committed certificates. The network contains no messages.

In the initial state, validators in the genesis committee can immediately create and broadcast certificates for round 1, via the first and second rules in Figure 2, since the *isClosed* and *isQuorum* checks on previous certificates do not apply in round 1. The certificates authored by correct validators at round 1 can be readily accepted by other correct validators via the third rule in Figure 2: the *isClosed* check does not apply in round 1, and the checks on signers are satisfied because they were already checked by the author according to the first rule. The certificates authored by faulty validators at round 1 are accepted by correct validators only if they pass the checks on signers: this is why accepting validators must independently perform those checks; faulty validators may create unacceptable certificates, e.g. with some signers outside the genesis committee or with not enough signers, but those certificates sit in the network forever.

Once a validator has received enough certificates at round 1, after advancing to round 2 via the fourth rule in Figure 2, the validator can use again the first and second rules in Figure 2 to create and broadcast a certificate at round 2. If the author is correct, the other correct validators can accept the certificate as soon as they have those in round 1 that it references, since again the checks on the signers have been performed by the author. If the author is faulty, correct validators independently check the signers, as in round 1.

This process can continue at rounds 3, 4, and so on, via suitable sequences of the first four rules in Figure 2. As DAGs are constructed, blocks may be generated. A validator may use the fifth rule at round 3, if all the conditions in the rule apply. A block may thus be generated for round 2, which might change the bonded committee at round 3. However, the genesis committee is still the active committee at rounds 3 through $lkbk + 2$, with $lkbk + 3$ being the earliest round at which the active committee may differ, which happens if the bonded committee at round 3 differs. If no block is generated for round 2, an attempt may

be made to generate a block for round 4, when the validator is at round 5. If that attempt is successful, it may also generate a block for round 2, if there is a path from the anchor for round 4 to the anchor for round 2.

An anchor may be skipped temporarily (eventually collected from a later anchor) or permanently. So long as a validator's blockchain does not lag too far behind, the validator can participate in later rounds, because the active committees for those rounds do not need the more recent blocks to be calculated, according to the lookback approach. If a blockchain lags too far behind, a validator may be unable to calculate the active committee at the current round, and may deadlock, potentially causing the protocol to deadlock. But deadlocks do not affect blockchain nonforking.[8]

The nondeterminism of our labeled state transition system lies in the choice of the next event to occur in a state. Once an event is chosen, the new state is uniquely determined: if $T(s, e, s')$ and $T(s, e, s'')$, then $s' = s''$.

3 Formal Proofs

This section presents the key aspects of our statement and proof of blockchain nonforking. Appendix B provides more details.

Blockchain nonforking and related properties are *state invariants*, i.e. predicates $P \subseteq S$ that hold on all the states that the system goes through, under the fault tolerance assumptions described shortly.

The general approach to verify an invariant P is to prove:

1. $s \in I \implies P(s)$, i.e. the invariant is established initially.
2. $s \in H \wedge T(s, e, s') \wedge P(s) \implies P(s')$, i.e. the invariant is preserved by every transition from reachable states.
3. $s \in H \wedge T^*(s, [\langle e_1, s_1 \rangle, \ldots, \langle e_n, s_n \rangle]) \wedge P(s) \implies P(s_n)$, i.e. the invariant is preserved by every execution from reachable states. This always follows from (2), by induction on n.
4. $s \in H \implies P(s)$, i.e. the invariant holds on all reachable states. This always follows from (1) and (3), since $I \subseteq H$.

However, blockchain nonforking and some other invariants only hold under *fault tolerance* assumptions. A *fault-tolerant state*, characterized by a predicate $bft \subseteq S$, is one where, for every active committee w calculated by any validator, the total stake of the faulty validators in w is at most $fstk(w)$, or equivalently the total stake of the correct validators in w is at least $qstk(w)$. As discussed in Section 2.6, this generalizes the fault tolerance condition of typical BFT systems, with $fstk(w)$ generalizing f and $qstk(w)$ generalizing $n-f$. The *fault-tolerant-reachable states*, characterized as a set $\widehat{H} \subseteq H$, are reachable states whose associated execution consists of fault-tolerant states. Invariants under fault tolerance are proved as in (1)–(4) above, but with H replaced by $\widehat{H}$ in (2)–(4), and with the additional hypothesis $s \in \widehat{H}$ in (1) since $I \not\subseteq \widehat{H}$ for every choice of *gcmt* (but this hypothesis is not actually needed to prove (1), since initial states in I have a very simple structure).

Some invariants *build upon* others, i.e. the latter are needed to prove the former. The main purpose of the H or $\widehat{H}$ hypotheses in (2) and (3) is to enable the use of previously proved invariants in those proofs—but it actually suffices to use those invariants as hypotheses, along with a *bft* hypothesis if needed. Invariants are organized and proved in a partial order, based on how they build upon each other. If the committee of validators were fixed, the elements of that partial order would be individual invariants, each of which could be completely proved on its own, after proving the invariants it builds upon; the last invariant proved would be blockchain nonforking. But with dynamic committees, certain invariants are *interdependent*, i.e. they must be proved together. For two such interdependent invariants P_1 and P_2, (1) is proved for each independently, but (2) is combined as $s \in H \wedge T(s, e, s') \wedge P_1(s) \wedge P_2(s) \implies P_1(s') \wedge P_2(s')$: that is, to prove that P_i holds on s', it must be assumed that not only P_i, but also P_j, holds on s. The same combination applies to (3), and (4) follows for both P_1 and P_2. The H is replaced with $\widehat{H}$ if needed. This generalizes to any number of interdependent invariants.

[8] In fact, deadlocked system states are easily reachable in our model: starting in an initial state, validators could immediately advance their rounds without creating certificates, getting stuck in round 2, waiting for certificates in round 1 that will never arrive. Our model is designed to capture a superset of the possible executions of an implementation.

Blockchain nonforking is expressed by saying that, if $\overline{b}_1$ and $\overline{b}_2$ are the blockchains of two validators in a fault-tolerant-reachable state $s \in \widehat{H}$, then $\exists\, \overline{b} : (\overline{b}_1 = \overline{b}_2 \bowtie \overline{b}) \vee (\overline{b}_2 = \overline{b}_1 \bowtie \overline{b})$. That is, either the two blockchains are equal (in which case $\overline{b} = \varnothing$), or one is a prefix of the other (in which case $\overline{b}$ is the extension).

Blockchain nonforking is a consequence of *anchor nonforking*, which is expressed similarly to blockchain nonforking above, but using the sequences of all the anchors committed by the validators, instead of their blockchains. As defined in Section 2, blockchains are generated piecewise from committed anchors, but it turns out that the piecewise generation yields the same result as recalculating the whole blockchain from the committed anchors; the $\overline{b}$ and $\widetilde{c}$ components of a validator state v are *redundant*, i.e. they can be calculated from other state components (for fault-tolerant-reachable system states).

Anchor nonforking is proved via a *successor-predecessor intersection* argument between (i) the total stake, in the DAG of a validator a_1, of the authors of the certificates at round r who elect an anchor c at round $r-1$ and (ii) the total stake, in the DAG of a validator a_2, of the authors of the certificates at round r referenced in the $\widetilde{p}$ component of a generic certificate c' at round $r+1$: that intersection is not empty, and therefore there is a path from c' to c in the DAG of a_2, which therefore must also contain c. The reason for requiring more than *fstk* voting stake is exactly so that it intersects the quorum *qstk*. The path extends to later rounds: every certificate in the DAG of a_2 at round $r+1$ or later has a path to the anchor c. This ensures that, given that a_1 commits the anchor c, a_2 will also commit it, if and when it commits some later anchor by electing it in its own DAG.

All of the above relies on *certificate nonequivocation*, i.e. that certificates in DAGs have unique author-round combinations: if a certificate c_1 in the DAG of a validator a_1 and a certificate c_2 in the DAG of a validator a_2 have the same author and round, i.e. $c_1.a = c_2.a$ and $c_1.r = c_2.r$, then they are the same certificate, i.e. $c_1 = c_2$. The case $a_1 = a_2$ is a special one, but it is crucial that nonequivocation holds across different validators. Certificate nonequivocation is proved via a *quorum intersection* argument between (i) the total stake of the signers of a generic existing certificate c in a DAG and (ii) the total stake of the signers needed to create or accept a certificate $c' \neq c$ with the same author and round. Under fault tolerance assumptions, the intersection must include at least one correct validator, which would not sign both c and c'; therefore, such a c' cannot be created or accepted.

However, the two quora being intersected may be based on two committees for the same round (of c and c') calculated by different validators from their respective blockchains. If the committees differed, the quorum intersection argument would fail. But the committees do not differ, because blockchains do not fork: if two validators can both calculate the committee for a round, then they calculate the same committee. Thus, certificate nonequivocation needs blockchain nonforking, which in turn needs certificate nonequivocation as mentioned above. This seeming circularity actually means that blockchain nonforking and certificate nonequivocation are interdependent invariants, along with other related invariants like *committee agreement*, which are all proved in a single induction.

The fault tolerance conditions explained above are the only assumptions needed in the proofs. There are no restrictions on how a committee can change: it is possible to replace a committee completely in a single block. Since the active committee for each round is consistently used to make decisions for that round as defined in Section 2, it does not matter if and how the active committees of two adjacent rounds differ.

4 Formal Model and Proofs in ACL2

The model in Section 2 and the proofs in Section 3 and Appendix B have been formalized in ACL2 [22], a general-purpose interactive theorem prover based on an untyped first-order classical logic of total functions that is an extension of a purely functional subset of Common Lisp [42]. The user interacts with ACL2 by submitting a sequence of theorems, function definitions, etc. ACL2 attempts to prove theorems automatically, via algorithms similar to NQTHM [11], most notably simplification and induction. The user guides these proof attempts mainly by (i) proving lemmas for use by specific proof algorithms (e.g. rewrite rules for the simplifier) and (ii) supplying theorem-specific 'hints' (e.g. to case-split on certain conditions).

The states and events in Figure 1 are formalized as algebraic data types, using an existing macro library to emulate such types in the untyped logic of ACL2 [43]. The initial states, as well as the auxiliary constants, functions, and relations in Figure 3, are formalized as ACL2 functions (nullary for constants, and boolean-valued for predicates). The transition rules in Figure 2 are formalized via (i) a predicate saying whether

an event is possible in a state, and (ii) a function mapping a state and an event to the new state when the predicate holds. Blockchain nonforking and other invariants are defined as ACL2 predicates, and theorems are proved saying that the predicates hold on every fault-tolerant-reachable state.

The ACL2 formalization, available open-source, consists of about 20,000 physical lines, including extensive documentation. Initially we formalized a version with static committees without stake, which already required fleshing out many key concepts and details, providing a solid foundation for tackling dynamic committees with stake. Extending the formalization to dynamic committees without stake involved a considerable jump in complexity, partly because of the interdependence of the invariants. The final extension to dynamic committees with stake required a smaller lift, but still several generalizations.

ACL2 verifies the whole formalization in less than 1 minute on an Apple M3 Max with 16 cores, and less than 2 minutes even using just a single core. While ACL2 usefully automates the low-level details, the proofs are organized to follow our lead, sparing the prover from exploring large search spaces. The proofs are carried out efficiently, and apply to arbitrarily long executions with arbitrarily large states.

We lead ACL2 to prove that blockchains never fork by defining blockchain nonforking as an invariant, fleshing out the other invariants and their relationships, and proving all the invariants by induction. Proving the establishment of invariants in initial states is generally automatic once ACL2 is instructed to expand some relevant function definitions. The same applies to the preservation of invariants by the events that do not affect those invariants, but the preservation proofs for the other events generally require user guidance, in the form of lemmas and hints. Given theorems for invariant establishment and preservation, the proofs by induction that invariants hold on reachable states are mostly automatic.

The ACL2 definition of the labeled state transition system is trusted to adequately capture the workings of the protocol, as relevant to the blockchain nonforking property, whose ACL2 definition is also trusted to express the right idea. Everything else, i.e. the other invariants and all the proofs, are not trusted (assuming that ACL2 is sound), being mere instruments to show that blockchains do not fork. However, the other invariants and all the proofs provide significant insights into how and why the protocol works.

5 Related Work

The most closely related work [9] proves safety of the DAG-based protocols DAG-Rider and Cordial Miners using TLA+ specifications and the TLAPS prover, using a compositional approach that is extensible to other protocols. The approach is limited to an arbitrary but fixed set of validators, and the composition is based on the layering of blockchain construction over DAG construction. However, our protocol has a dynamic set of validators and has intertwined DAG and blockchain construction, making that approach inapplicable.

An earlier work [15] proves the correctness of the DAG-based protocol Hashgraph in Coq, and mentions some (fixable) errors in the original proof. The proof assumes a fixed set of validators.

Properties of non-DAG-based protocols have also been formally verified, including Stellar using LTL, IVy, and Isabelle/HOL [28], Tendermint using IVy [27] Pipelined Moonshot using IVy [33], Algorand using Coq [4], LibraBFT using Agda [12], and Red Belly using ByMC [10]. Although there are commonalities with our work (e.g. quorum intersection arguments), the protocols work differently and require different techniques.

Velisarios [34] and Bythos [44] are frameworks for specifying and verifying BFT protocols in Coq. It would be interesting to try to verify our protocol in these frameworks, which, to our knowledge, have not yet been used for DAG-based protocols.

The DistAlgo [25, 26] language is an extension of Python with a formal operational semantics. It has been used to implement consensus protocols and can be translated to TLA+ for verification.

6 Conclusion

We have formalized, in the ACL2 theorem prover, a model of a DAG-based BFT consensus protocol with dynamic stake, and a proof that blockchains never fork, under standard fault tolerance assumptions on the dynamic validator sets. The fact that the set of validators can change at every block makes the proofs more challenging because DAG construction and blockchain construction are not layered as in the case of a fixed set of validators. Our nonforking proofs apply to arbitrarily long executions with arbitrarily large system states; the model and proofs are verified in 1 minute by ACL2. This work not only shows that blockchains

using this protocol do not fork (which was unclear prior to this work), but also elucidates how the classical fault tolerance assumptions generalize to this form of dynamic sets of validators with stake. Our work also provides a solid foundation for studying additional features and properties of this kind of protocol, such as syncing and liveness.

A Mathematical Notation

We use $\wedge$ for conjunction, $\vee$ for disjunction, $\implies$ for implication, $\impliedby$ for reverse implication (the right side implies the left side), $\iff$ for bi-implication (logical equivalence), $\forall$ for universal quantification, $\exists$ for existential quantification, and $\nexists$ for negated existential quantification (i.e. "there does not exist").

We use $\varnothing$ for the empty set, $\in$ for set membership, $\notin$ for set non-membership, $\subseteq$ for (non-strict) subset, $\cup$ for set union, $\cap$ for set intersection, $\setminus$ for set difference, and $|\ldots|$ for set cardinality. We construct sets by listing their elements, e.g. $\{0\}$ in Figure 1, or using a set comprehension, e.g. $\{f \in K^0 \mid f < nstk(w)/3\}$ in Figure 3.

If $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are sets, with $n \geq 2$, then $\mathcal{S}_1 \times \cdots \times \mathcal{S}_n$ is their Cartesian product, i.e. the set of all n-tuples $\langle \alpha_1, \ldots, \alpha_n \rangle$, where each $\alpha_i \in \mathcal{S}_i$.

If $\mathcal{S}$ is a set, $\widetilde{\mathcal{S}}$ is the set of all finite subsets of $\mathcal{S}$. For example, $\widetilde{M}$ in Figure 1 is the set of all finite sets of messages from M.

If $\mathcal{S}$ is a set, $\overline{\mathcal{S}}$ is the set of all finite sequences $[\alpha_1, \ldots, \alpha_n]$, where each $\alpha_i \in \mathcal{S}$. We use ϵ for the empty sequence. We use $\bowtie$ to concatenate sequences. For example, $\overline{B}$ in Figure 1 is the set of all finite sequences of blocks from B, including all possible blockchains.

If $\mathcal{S}$ and $\mathcal{S}'$ are sets, $\mathcal{S} \to \mathcal{S}'$ is the set of total functions from $\mathcal{S}$ to $\mathcal{S}'$. As customary, we write $\varphi : \mathcal{S} \to \mathcal{S}'$ for $\varphi \in \mathcal{S} \to \mathcal{S}'$.

If $\mathcal{S}$ and $\mathcal{S}'$ are sets, $\mathcal{S} \rightsquigarrow \mathcal{S}'$ is the set of finite maps from $\mathcal{S}$ to $\mathcal{S}'$, i.e. total functions from finite subsets of $\mathcal{S}$ to $\mathcal{S}'$. The domain of a finite map μ is $\mathcal{D}(\mu)$. For example, $A \rightsquigarrow V$ in Figure 1 is the set of all finite maps from addresses to validator states.

Since a finite map is a function, $\mu(\alpha)$ is the value of μ at α, if $\alpha \in \mathcal{D}(\mu)$. For example, $\vec{v}(a)$ is the validator state at address a in the validator map $\vec{v}$, used in several places in Figure 2.

If μ is a finite map, $\mu\{\alpha \mapsto \beta\}$ is the finite map that is like μ except that it maps α to β: if $\alpha \in \mathcal{D}(\mu)$, then β replaces $\mu(\alpha)$ in the new map; if $\alpha \notin \mathcal{D}(\mu)$, the new map has a domain extended with α. For example, $\vec{v}\{a \mapsto v'\}$ is the validator map that is the same as $\vec{v}$ except that it maps a to v', used in several places in Figure 2.

If μ is a finite map and $\mathcal{S} \subseteq \mathcal{D}(\mu)$, $\mu{\downarrow_{\mathcal{S}}}$ is the restriction of μ to $\mathcal{S} \subseteq \mathcal{D}(\mu)$, i.e. the finite map whose domain is $\mathcal{S}$ and that maps each $\alpha \in \mathcal{S}$ to $\mu(\alpha)$. For example, $w{\downarrow_{\mathcal{D}(w)\setminus\{a\}}}$ in the definition of *cmt* in Figure 3 is the committee that is the same as w except that it does not have an entry for a in its domain.

A relation or predicate over a set $\mathcal{S}$ is a subset $\rho \subseteq \mathcal{S}$. We often use a functional notation $\rho(\alpha)$ for $\alpha \in \rho$, as if it were a boolean-valued function. For example, for the transition relation T of the labeled state transition system defined in Section 2.1, $T(s, e, s')$ stands for $\langle s, e, s' \rangle \in T$.

We use the following lexical conventions. Sets that represent "types" of entities, such as A, R, and other sets defined in Figure 1, are denoted by uppercase single letters, possibly with subscript and superscript decorations as in X_{B} and R^0. Elements of such sets are often denoted by lowercase single letters corresponding to the uppercase set names, e.g. $a \in A$ and $c \in C$; when more than such variable is needed, we use tick marks to distinguish them, e.g. v, v', and v'' are three variables ranging over V. Elements of sets of finite sets like $\widetilde{M}$ or of sets of finite sequences like $\overline{B}$ are often decorated with a tilde or overline, e.g. $\widetilde{m} \in \widetilde{M}$ and $\overline{b} \in \overline{B}$, matching the notation that constructs these sets; however, the tilde or overline in $\widetilde{m}$ and $\overline{b}$ are just decorations, not set operators. We use an arrow decoration in the finite map $\vec{v}$ from addresses to validator states, to distinguish it from validator states $v \in V$. Constants, functions, and relations in Figure 3 and Figure 8 have multi-letter names, mostly in lowercase. Multi-letter symbols in small uppercase, like BOND and ACCEPT, are used as symbolic "tags". The symbol $\perp$ is used as a value in a few function definitions to indicate that the function is conceptually "undefined" for the given arguments.

When we define a cartesian product in Figure 1, we simultaneously define notations for component access. For example, the line

$$c = \langle a, r, \overline{x}, \widetilde{p}, \widetilde{q} \rangle \in C = A \times R \times \overline{X} \times \widetilde{A} \times \widetilde{A}$$

not only defines the structure of a certificate, but also defines the accessors $.a$, ..., $.\widetilde{q}$ For example, $c'.\overline{x}$ is the transaction sequence component of certificate c'. To create a tuple that is a copy of another tuple with one component changed, we use the notation $\mathit{oldtuple}\langle \mathit{component} \mapsto \mathit{newvalue} \rangle$. For example, $v\langle g \mapsto v.g \cup \{c\} \rangle$ is the validator state that is the same as v except that it has the certificate c added to its DAG (if not already in the DAG).

Last block round:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge v = s.\vec{v}(a) \Longrightarrow v.l = \mathit{last}(v.\overline{b})$
Ordered block rounds:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge s.\vec{v}(a).\overline{b} = [b_1, \ldots, b_n] \Longrightarrow b_1.r < \cdots < b_n.r$
Even block rounds:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge s.\vec{v}(a).\overline{b} = [b_1, \ldots, b_n] \wedge 1 \leq i \leq n \Longrightarrow b_i.r \bmod 2 = 0$
Backward closure:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge g = s.\vec{v}(a).g \wedge c \in g \wedge p \in c.\widetilde{p} \Longrightarrow \exists\, c' \in g : (c'.a = p) \wedge (c'.r = c.r - 1)$
Signer quorum:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge c \in s.\vec{v}(a).g \Longrightarrow \mathit{isQuorum}(\{c.a\} \cup c.\widetilde{q}, c.r, s.\vec{v}(a).\overline{b})$
Signer records:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge c \in \mathit{signedCerts}(a, s) \wedge v = s.\vec{v}(a) \Longrightarrow$ $(\exists\, c' \in v.g : c'.a = c.a \wedge c'.r = c.r) \vee \langle c.a, c.r \rangle \in v.\widetilde{d}$
No self-endorsement:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \Longrightarrow \nexists\, r : \langle a, r \rangle \in s.\vec{v}(a).\widetilde{d}$
Signed nonequivocation:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge c_1, c_2 \in \mathit{signedCerts}(a, s) \wedge c_1.a = c_2.a \wedge c_1.r = c_2.r \Longrightarrow c_1 = c_2$
DAG nonequivocation:	$s \in \widehat{H} \wedge a_1, a_2 \in \mathcal{D}(s.\vec{v}) \wedge c_1 \in s.\vec{v}(a_1).g \wedge c_2 \in s.\vec{v}(a_2).g \wedge c_1.a = c_2.a \wedge c_1.r = c_2.r \Longrightarrow c_1 = c_2$
Signed previous quorum:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge c \in \mathit{signedCerts}(a, s) \Longrightarrow$ $(c.r = 1 \wedge c.\widetilde{p} = \varnothing) \vee (c.r \neq 1 \wedge c.\widetilde{p} \neq \varnothing \wedge \mathit{isQuorum}(c.\widetilde{p}, c.r - 1, s.\vec{v}(a).\overline{b}))$
DAG previous quorum:	$s \in \widehat{H} \wedge a \in \mathcal{D}(s.\vec{v}) \wedge c \in s.\vec{v}(a).g \Longrightarrow$ $(c.r = 1 \wedge c.\widetilde{p} = \varnothing) \vee (c.r \neq 1 \wedge c.\widetilde{p} \neq \varnothing \wedge \mathit{isQuorum}(c.\widetilde{p}, c.r - 1, s.\vec{v}(a).\overline{b}))$
Last anchor presence:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge v = s.\vec{v}(a) \wedge v.l \neq 0 \Longrightarrow \exists\, c : \mathit{isLastAnch}(c, v)$
Last anchor voters:	$s \in H \wedge a \in \mathcal{D}(s.\vec{v}) \wedge v = s.\vec{v}(a) \wedge \mathit{isLastAnch}(c, v) \Longrightarrow \mathit{isElected}(c, v.g, v.\overline{b})$
Anchor paths:	$s \in \widehat{H} \wedge a, a' \in \mathcal{D}(s.\vec{v}) \wedge \mathit{isLastAnch}(c, s.\vec{v}(a)) \wedge c' \in s.\vec{v}(a').g \wedge c'.r \geq c.r + 2 \Longrightarrow \mathit{isPath}(c', c, s.\vec{v}(a').g)$
Anchor nonforking:	$s \in \widehat{H} \wedge a_1, a_2 \in \mathcal{D}(s.\vec{v}) \wedge \overline{c}_1 = \mathit{collAllAnch}(s.\vec{v}(a_1)) \wedge \overline{c}_2 = \mathit{collAllAnch}(s.\vec{v}(a_2)) \Longrightarrow$ $\exists\, \overline{c} : (\overline{c}_1 = \overline{c}_2 \bowtie \overline{c}) \vee (\overline{c}_2 = \overline{c}_1 \bowtie \overline{c})$
Committed redundancy:	$s \in \widehat{H} \wedge a \in \mathcal{D}(s.\vec{v}) \wedge v = s.\vec{v}(a) \Longrightarrow$ $((\nexists\, c : \mathit{isLastAnch}(c, v)) \wedge v.\widetilde{c} = \varnothing) \vee (\exists\, c : \mathit{isLastAnch}(c, v) \wedge v.\widetilde{c} = \{c' \mid \mathit{isPath}(c, c', v.g)\})$
Blockchain redundancy:	$s \in \widehat{H} \wedge a \in \mathcal{D}(s.\vec{v}) \wedge v = s.\vec{v}(a) \wedge \mathit{genBlk}(\mathit{collAllAnch}(v), v.g, \epsilon, \varnothing) = \langle \overline{b}, \widetilde{c} \rangle \Longrightarrow v.\overline{b} = \overline{b}$
Blockchain nonforking:	$s \in \widehat{H} \wedge a_1, a_2 \in \mathcal{D}(s.\vec{v}) \wedge \overline{b}_1 = s.\vec{v}(a_1).\overline{b} \wedge \overline{b}_2 = s.\vec{v}(a_2).\overline{b} \Longrightarrow \exists\, \overline{b} : (\overline{b}_1 = \overline{b}_2 \bowtie \overline{b}) \vee (\overline{b}_2 = \overline{b}_1 \bowtie \overline{b})$
Committee agreement:	$s \in \widehat{H} \wedge a_1, a_2 \in \mathcal{D}(s.\vec{v}) \wedge r \in R \wedge w_1 = \mathit{acmt}(r, s.\vec{v}(a_1).\overline{b}) \neq \bot \wedge w_2 = \mathit{acmt}(r, s.\vec{v}(a_2).\overline{b}) \neq \bot \Longrightarrow w_1 = w_2$

Figure 7: Invariants

B Invariants and Proof Sketches

Figure 7 defines blockchain nonforking and all the other invariants that we have proved. Their definitions make use of the additional auxiliary sets, functions, and relations in Figure 8. The rest of this appendix describes the invariants and sketches their proofs.

The proofs generally follow the approach outlined in Section 3, with steps (1)–(4) for each invariant. However, a few invariants are, and can be proved as, *direct consequences* of others: for such an invariant P, it suffices to prove an implication of the form $P_1(s) \wedge P_2(s) \wedge \cdots \Longrightarrow P(s)$; then (4) for P follows from (4) for P_1, P_2, etc. If fault tolerance is needed for these direct-consequence invariants, the $\mathit{bft}(s)$ hypothesis is added to the implication.

B.1 Fault Tolerance

The predicate *bft* in Figure 8 says that a state is fault-tolerant exactly when every active committees w at any round r calculated by any correct validator a from its blockchain, is such that the total stake of the faulty validators in w does not exceed the maximum faulty stake $\mathit{fstk}(w)$. This is equivalent to saying that the total stake of the correct validators in w is at least the quorum stake $\mathit{qstk}(w)$.

The set $\widehat{H}$ of fault-tolerant-reachable states is defined in Figure 8 to consist of all the reachable states such that all the states that the execution goes through are fault-tolerant. Note that $\widehat{H} \subseteq H$.

Fault-tolerant states: $bft \subseteq S \qquad bft(s) \iff \forall\, a \in \mathcal{D}(s.\vec{v}), r \in R, w \in W : (w = acmt(r, s.\vec{v}(a).\bar{b}) \implies \sum_{a' \in \mathcal{D}(w) \setminus \mathcal{D}(s.\vec{v})} w(a') \leq fstk(w))$

Fault-tolerant-reachable states: $\widehat{H} = \{s_n \in S \mid \exists\, e_1, \ldots, e_n, s_1, \ldots, s_{n-1} : T^*(s_0, [\langle e_1, s_1\rangle, \ldots, \langle e_n, s_n\rangle]) \wedge bft(s_0) \wedge \cdots \wedge bft(s_n)\}$

Certificates in system: $allCerts : S \to \widetilde{C} \qquad allCerts(s) = \bigcup_{a \in \mathcal{D}(s.\vec{v})} s.\vec{v}(a).g \cup \{m.c \mid m \in s.\widetilde{m}\}$

Certificates signed by validator: $signedCerts : A \times S \to \widetilde{C} \qquad signedCerts(a, s) = \{c \in allCerts(s) \mid a = c.a \vee a \in c.\widetilde{q}\}$

Last committed anchor: $isLastAnch \subseteq C \times V \qquad isLastAnch(c,v) \iff isAnchor(c, v.g, v.\bar{b}) \wedge c.r = v.l$

All committed anchors: $collAllAnch : V \to \overline{C}$ $\begin{cases} collAllAnch(v) = collAnch(c, 0, v.g, v.\bar{b}) & \Longleftarrow\ isLastAnch(c,v) \\ collAllAnch(v) = \epsilon & \Longleftarrow\ \nexists\, c : isLastAnch(c,v) \end{cases}$

Figure 8: Additional Auxiliary Sets, Functions, and Relations

B.2 Initial States

All the invariants in Figure 7 trivially hold in the initial states, i.e. when $s \in I$. This is because, as defined in I in Figure 1, all DAGs, blockchains, etc. are empty. This covers (1) in Section 3.

The interesting proofs are the ones for the preservation of the invariants by transitions, i.e. (2) in Section 3. The sketches given below are for these preservation proofs.

B.3 Block Rounds

The first three invariants in Figure 7 concern the block rounds in each validator's blockchain: the last committed round is the one of the latest block (or 0 if there are no blocks; see *last* in Figure 3); the block rounds are even, and strictly increase from left to right.

Blockchains only change via COMMIT events, but in a way that preserves these invariants, as easily seen from the transition rules in Figure 2 and the functions in Figure 3 used by the rules. But the preservation proof of the second invariant needs the first invariant as hypothesis: if $v.l$ were below $last(v.\bar{b})$, new blocks could be generated for already committed anchors.

The significance of the second and third invariants is that the calculation of (bonded, and therefore active) committees is "monotonic" under blockchain extension: extending the blockchain enables the calculation of committees at more rounds, but does not change the committees calculable before the extension. This is because *bcmt* and *bcmt′*, and therefore *acmt*, in Figure 3, are implicitly based on the block numbers satisfying those invariants.

B.4 Backward Closure

The backward closure invariant in Figure 7 says that, for every certificate c in a validator's DAG, the DAG has all the certificates referenced by c in the previous round (i.e. no "dangling edges"). If $c.r = 1$, there is no $p \in c.\widetilde{p}$, and the invariant trivially holds.

DAGs only change via CREATE and ACCEPT events, whose transition rules add a certificate to the DAG only if its previous certificates are already in the DAG. Certificates are never removed from DAGs.

This is a critical property, which provides, together with the nonequivocation invariant discussed later, a certain "stability" under changes to DAGs. Adding a certificate to a DAG does not alter the presence of absence of paths between existing certificates; it can only add paths from the added certificate. Intuitively, this means that it makes no difference whether an anchor is committed sooner rather than later (e.g. whether it has enough votes itself, or it is reachable from a later anchor with enough votes): the same block is generated from the anchor. Furthermore, if *collAnch* in Figure 3 skips an anchor at some point (because the anchor is absent or not reachable from a committed one), it would have skipped it as well even if the collection had taken place later, with more certificates in the DAG.

B.5 Signer Quorum

The signer quorum invariant in Figure 7 says that, for every certificate c in a validator's DAG, the validator can calculate the active committee at the round of c, and the signers of the certificate form a quorum in that committee.

DAGs only change via CREATE and ACCEPT events, whose transition rules add a certificate to the DAG only if *isQuorum* holds. Although COMMIT events do no change DAGs, they change the blockchain, which *isQuorum* depends on. However, the active committee for the round of c does not change, because of the block round invariants in Section B.3 (as discussed there).

The signer quorum invariant is used to prove other invariants below, specifically the ones based on quorum intersection.

B.6 Signer Records and No Self-Endorsement

The signer records invariant in Figure 7 says that every signer of every certificate in the system has a "record" of (the author and round of) the certificate. The *signedCerts* function in Figure 8 returns the set of certificates signed by a given validator, out of all the certificates in the system state returned by *allCerts* in Figure 8: the latter consist of the certificates in all the DAGs and in the network, from which *signedCerts* selects the ones with a given author or endorser. The invariant says that, for each certificate c signed by a, the author and round of c are in the DAG or in the set of endorsed pairs. There is a subtle technical reason why the first disjunct is not just that c is in the DAG: although that is always the case because of DAG nonequivocation, this fact is not available in the proof of the signer records invariant, which is in fact used to prove DAG nonequivocation.[9]

When a CREATE event creates a certificate authored by a correct validator, the certificate is added to the author's DAG, so the first disjunct holds for the author; when a CREATE event creates a certificate authored by any validator, the certificate's author and round are added the every correct endorser's set of endorsed pairs, so the second disjunct holds for the endorsers. When an ACCEPT event adds a certificate c to an endorser's DAG, the author-round pair is removed from the set of endorsed pairs, so the second disjunct no longer holds; but c is added to the DAG, so the first disjunct holds. The other events do not modify DAGs and sets of endorsed pairs.

The no self-endorsement invariant in Figure 7 says that each validator's set of endorsed pairs does not contain any pair with the validator as author. This is because validators only endorse certificates authored by others. When a CREATE event creates a certificate authored by a correct validator, the author is distinct from the endorsers, so the new endorsed pairs satisfy the invariant. An ACCEPT event may remove a pair, which preserves the invariant. The other events do not modify the sets of endorsed pairs.

The no self-endorsement invariant "refines" the signer records invariant by restricting the record held by an author. The significance of these two invariants is that, by keeping and using these records, a correct validator never signs equivocal certificates, as described next.

B.7 Signed Nonequivocation

The signed nonequivocation invariant in Figure 7 says that the certificates signed by a correct validator are unequivocal: if two such certificates have the same author and round, they are equal.

Only an CREATE event adds a certificate to *signedCerts*, but the two rules in Figure 2 ensure that the correct signers do not already have records of the new certificates's author and round. Endorsers do so via *isNew*, which is the exact negation of the signer records invariant. Authors only check their DAG, but the no self-endorsement invariant implies that the additional check on endorsed pairs is not needed. Thus, since the invariants in Section B.6 hold, no CREATE event can add an equivocal certificates to *signedCerts*.

This invariant provides a key fact to prove DAG nonequivocation, as described next.

B.8 DAG Nonequivocation

The DAG nonequivocation invariant in Figure 7 says that if two certificates in two validators' DAGs have the same author and round, they are equal. The two validators may differ or not; if $a_1 = a_2$, the invariant concerns a single DAG as a special case. This DAG agreement invariant is a major component of blockchain consensus, since blockchains are derived from DAGs.

[9] Although it might be possible to make this invariant interdependent with DAG nonequivocation and others, and strengthen the first disjunct to say that c is in the DAG, the weaker form is sufficient to prove the signer record invariant on its own and to use it to prove other invariants including DAG nonequivocation.

The proof is based on a *quorum intersection* argument, which is a common technique in BFT systems. At a high level, the argument is as follows, using the typical static validator counts n and f for simplicity (see the discussion at the end of Section 2.6). Suppose that there were two equivocal certificates in DAGs, i.e. two certificates $c_1 \neq c_2$ with $c_1.a = c_2.a$ and $c_1.r = c_2.r$. By the signer quorum invariant in Section B.5, each certificate is signed by $n - f$ validators: let $\widetilde{a}_1$ and $\widetilde{a}_2$ be the sets of signers of c_1 and c_2, with $|\widetilde{a}_1| = |\widetilde{a}_2| = n - f$. Since $|\widetilde{a}_1 \cup \widetilde{a}_2| \leq n$, and since $|\widetilde{a}_1 \cup \widetilde{a}_2| = |\widetilde{a}_1| + |\widetilde{a}_2| - |\widetilde{a}_1 \cap \widetilde{a}_2|$, we have $|\widetilde{a}_1 \cap \widetilde{a}_2| \geq f + 1$, i.e. at least one correct validator must have signed both certificates, assuming fault tolerance. But by the signed nonequivocation invariant in Section B.7, this is impossible, and thus the hypothesized equivocal certificates cannot exist.

The above argument with static counts n and f extends to stake: the intersection of the signers has more than the total assumed stake of faulty validators, and thus at least one signer is correct. But the extension to dynamic committees involves a significant complication: if c_1 and c_2 are from different DAGs, the two validators might calculate different committees for the common round, defeating the quorum intersection argument. DAG nonequivocation needs the two validators to agree on the committees, which needs the two validators to agree on their blockchains, which needs DAG nonequivocation. This seeming circularity is resolved by proving DAG nonequivocation, committee agreement, blockchain nonforking, and other related invariants, simultaneously by induction, as interdependent invariants as discussed in Section 3. When proving that DAG nonequivocation is preserved by transitions, the committee agreement invariant at the end of Figure 7 can be assumed as hypothesis on the old state, which makes the quorum intersection argument work, showing that DAG nonequivocation holds on the new state as well.

Our actual proof is not quite by contradiction as above, hypothesizing that both c_1 and c_2 were in DAGs. Instead we show that, if c_1 is in a DAG, a CREATE or ACCEPT event could not add c_2 to a DAG (another DAG or the same DAG): the signer quorum for c_1 is derived from the signer quorum invariant, but the signer quorum for c_2 is derived from the conditions under which the event can happen.

Unlike the invariants discussed before, DAG nonequivocation only holds on fault-tolerant-reachable states. Fault tolerance is needed for the quorum intersection argument.

B.9 Signed Previous Quorum

The signed previous quorum invariant in Figure 7 says that, for each certificate c signed by a correct validator a, either the round of c is 1 and c has no references to previous certificates, or the round of c is not 1 and the references to previous certificates in c form a non-empty quorum in the active committee for the previous round, as calculated by a using its blockchain.

Only CREATE events add certificates to *signedCerts*, but they do so only if author and endorser satisfy this condition. The other events do not add certificates, but COMMIT events extend blockchains, which *isQuorum* depends on. However, the active committee for the round of c does not change, because of the block round invariants in Section B.3 (as discussed there).

B.10 DAG Previous Quorum

The DAG previous quorum invariant in Figure 7 says the same thing as the signed previous quorum invariant, but for the DAG of a validator instead of the set of certificates signed by a validator.

Only CREATE and ACCEPT events add certificates to DAGs. When a CREATE event adds a certificate to the author's DAG, the first rule in Figure 2 explicitly ensures that the invariant holds on the new certificate. When an ACCEPT event adds a certificate c to the DAG of a validator a, there is no such explicit check; however, there is a signer quorum check, which, assuming fault tolerance, ensures that at least one signer a' is correct: with the typical static validator counts n and f for simplicity, $n - f \geq f + 1$, and that generalizes to dynamic stake. Since a' is correct, and $c \in \mathit{signedCerts}(a', s)$, the signed previous quorum invariant ensures that the desired condition holds on $c.\widetilde{p}$, but with respect to the committee calculated by a' instead of the validator a that has the DAG. Using the committee agreement invariant at the end of Figure 7, which is therefore interdependent with the DAG previous quorum invariant, ensures that a and a' calculate the same committee.

This invariant is used in an intersection argument (different from quorum intersection), described later.

B.11 Last Anchor Presence and Voters

The last anchor presence invariant in Figure 7 says that if a validator has committeed at least one anchor, then it has a last anchor, as formalized by *isLastAnch* in Figure 8. The predicate says that there is an anchor (see *isAnchor* in Figure 3) at the last committed round in a validator state.

Only COMMIT events change the last committed round, but they do so only if there is an anchor at the last committed round, because that is how the last committed round is determined. The event also extends the blockchain, which affects *isAnchor* and thus *isLastAnch*, but the block round invariants in Section B.3 ensure (as discussed there) that the blockchain extension does not change the active committee at the round relevant to *isLastAnch*.

The last anchor voters invariant in Figure 7 says that if a validator has a last committed anchor, then the certificates at the following round that vote for the anchor have authors whose total stake is above *fstk*, as expressed by *isElected* in Figure 3.

The required condition is explicitly checked in the transition rule for COMMIT events in Figure 2; the extension of the blockchain does not affect the active committee at the round of interest, because of the block round invariants in Section B.3 (as discussed there). A CREATE or ACCEPT event may add a voter for the anchor, increasing the voters' stake, which therefore still satisfies the inequality. The other events do not affect the references to the last anchor in the certificates at the following round.

The last anchor presence invariant enables the use of the last anchor voters invariant in the intersection argument below.

B.12 Anchor Paths

The anchor paths invariant in Figure 7 says that, if a validator a has a last committed anchor c at round r, and a validator a' has a certificate c' at round $r+2$ or later, then there is a path, in the DAG of a', from c' to c, which implies that the anchor c in the DAG of a is also present in the DAG of a'. This is a strong and important property for consensus: if any validator a commits an anchor c, either elected or reachable from an elected one (see Section 2.10), any other validator a' commits it as well, if and when it commits some later anchor, since every certificate at least two rounds after c, and in particular every anchor at least two rounds after c, has a path to c.

The proof of this invariant relies on an intersection argument, between (i) the set $\widetilde{a}$ of the authors of the certificates at round $r+1$ in the DAG of a that have edges to (i.e. vote for) c, and (ii) the set $\widetilde{a}'$ of the authors of the certificates at round $r+1$ in the DAG of a' that have edges from (i.e. are predecessors of) a generic certificate c' at round $r+2$. Using again the typical static validator counts n and f for simplicity, the last anchor voters invariant in Section B.11 implies that $|\widetilde{a}| > f$, and the DAG previous quorum invariant in Section B.10 implies that $|\widetilde{a}'| \geq n - f$. Since $|\widetilde{a} \cup \widetilde{a}'| \leq n$, and since $|\widetilde{a} \cup \widetilde{a}'| = |\widetilde{a}| + |\widetilde{a}'| - |\widetilde{a} \cap \widetilde{a}'|$, we have $|\widetilde{a} \cap \widetilde{a}'| > 0$, i.e. there is at least one validator a_0 in both sets. This intersection argument is similar to Section B.8, but instead of involving two quora of signers, it involves a quorum of predecessors and a set of voters, where the latter's size is chosen, in the protocol, exactly so that it intersects a quorum. This intersection argument, unlike the quorum intersection argument, does not involve correct or faulty validators, and does not need any fault tolerance assumption. This intersection argument generalizes to dynamic stake; similarly to Section B.8, since a and a' may differ, we need to assume committee agreement, making this invariant interdependent with that and others.

The existence of an address a_0 in the intersection, together with the backward closure invariant in Section B.4, implies that both the DAG of a and the DAG of a' have a certificate c_0 at round $r+1$ authored by the common validator a_0, that c is also in the DAG of a', and that there is a path from c' to c via c_0. This holds for every c' at round $r+2$ in the DAG of a'. Since every c'' at round $r+3$ in the DAG of a' has at least a previous certificate c' at round $r+2$, there is also a path from c'' to c. And so on for certificates at all the later rounds in the DAG of a'.

B.13 Anchor Nonforking

The function *collAllAnch* in Figure 8 returns all the anchors committed in a validator state, using *collAnch* in Figure 3 from the last committed anchor all the way to the start of the DAG, or returning the empty sequence if no anchor has been committed yet. The anchor nonforking invariant in Figure 7 says that the

sequences $\overline{c}_1$ and $\overline{c}_2$ of anchors committed by any two validators do not fork: either they are equal (in which case $\overline{c} = \epsilon$), or one extends the other (in which case $\overline{c} \neq \epsilon$ is the extension).

Given the two validator states v_1 and v_2, there are three cases to consider, based on whether $v_1.l = v_2.l$, or $v_1.l < v_2.l$, or $v_1.l > v_2.l$. The last two cases are symmetric, so considering one suffices. When $v_1.l = 0$ or $v_2.l = 0$, the proof is easy because the corresponding anchor sequence is empty.

If $v_1.l = v_2.l \neq 0$, assuming the interdependent committee agreement invariant, the two validators have the same leader at that round, and assuming the interdependent DAG nonequivocation invariant, the last committed anchors are the same, and so are all the anchors collected from them. So $\overline{c}_1 = \overline{c}_2$.

If $v_1.l < v_2.l$, the anchor paths invariant implies that the last committed anchor in v_1 is also in v_2, and there is a path to it from the last committed anchor in v_2. Thus, when calculating $\overline{c}_2$ from the last committed anchor of v_2, we collect one or more anchors $\overline{c}$, until we hit the last committed anchor of v_1, at which point we collect the same anchors $\overline{c}_1$ as in v_1, because of the interdependent DAG nonequivocation invariant. That is, $\overline{c}_2 = \overline{c}_1 \bowtie \overline{c}$.

Since blocks are generated from committed anchors, one per anchor, the nonforking of anchors is a key step towards proving the nonforking of blockchains. The invariants below describe how that proof is completed.

B.14 Committed Redundancy

The committed redundancy invariant in Figure 7 says that the $v.\widetilde{c}$ component of a validator state is redundant, i.e. it can be calculated from other state components: if there is no last committed anchor, the set is empty; if there is a last committed anchor, the set is the anchor's causal history.

A CREATE or ACCEPT event may add a certificate to a DAG, but that does not affect the causal history of any existing certificate (and in particular the last committed anchor). Because of the backward closure invariant, there are no "dangling edges" that could be filled and augment the causal history. And because of the DAG nonequivocation invariant, the added certificate cannot "overlap" with an existing one because it must have distinct author or round.

A COMMIT event does not change the DAG, but changes the last committed anchor, as well the set of committed certificates by adding the causal history of the new anchor. Because of the anchor paths invariant, the new anchor has a path to the old one, and thus its causal history is a superset of the old one, so the union is equal to the new causal history.

This invariant is interdependent with DAG nonequivocation and others, and thus it only holds on fault-tolerant-reachable states.

Since the $v.\widetilde{c}$ state component is redundant, we could have omitted it from our model in Section 2 altogether. However, the reasons for this redundancy are nontrivial, depending on many invariants. Furthermore, including that state component and its handling makes the model closer to an implementation.

B.15 Blockchain Redundancy

The blockchain redundancy invariant in Figure 7 says that the blockchain can be calculated from the sequence of committed anchors. From the rule for COMMIT in Figure 2, it is clear that the blockchain is created from the committed anchors, but in a piecewise manner; this invariant says that it can be entirely reconstructed from them.

This invariant holds because the function *genBlk* in Figure 3, which is applied to *collAllAnch* in Figure 8 to calculate the complete blockchain, satisfies a compositionality property on sequence concatenation, under conditions implied by DAG nonequivocation and other invariants: when a COMMIT event commits a new anchor, the new value of *collAllAnch* is the concatenation of its old value and the newly collected anchors. This compositionality property transfers to the blockchain calculation, relying on the fact that the certificates whose transactions go into the blocks are also redundantly determined from the committed anchors, because of the invariant in Section B.14. Other events do not change the blockchain; CREATE and ACCEPT events add certificates to DAGs, but in a way that does not affect existing causal histories, as discussed for other invariants.

This invariant is interdependent with DAG nonequivocation and others, and thus it only holds on fault-tolerant-reachable states.

The observation made at the end of Section B.14 about modeling the redundant $v.\widetilde{c}$ state component applies to $v.\overline{b}$ as well.

B.16 Blockchain Nonforking

The blockchain nonforking invariant in Figure 7 says that the blockchains $\overline{b}_1$ and $\overline{b}_2$ of any two validators do not fork: either they are equal (in which case $\overline{b} = \epsilon$), or one extends the other (in which case $\overline{b} \neq \epsilon$ is the extension). This has the same form as the anchor nonforking invariant, but with block sequences instead of anchor sequences.

Intuitively, this invariant holds because committed anchors do not fork (see Section B.13), and blockchains are calculated from those committed anchors (see Section B.15). But the proof involves a technical subtlety. Although it would be possible to prove this invariant as directly implied from the other invariants, that would lead to a circularity that induction cannot untangle. Instead, we prove that the blockchain nonforking invariant is preserved by each transition, using the previously proved transition preservation of the anchor nonforking invariant, and the direct implication in the new state of the transition. This lets the induction resolve the circularity: in the step case of the induction, all the interdependent invariants are assumed in the old state as induction hypothesis, and all of them are proved, one after the other, to hold in the new state.

Blockchain nonforking is the top-level invariant, in the sense that it is the main consensus property of the protocol, as stated in the title of this paper. However, as explained, it is at the same "level" as other interdependent invariants in the proof structure.

B.17 Committee Agreement

The committee agreement invariant in Figure 7 says that if two validators can both calculate the active committee at a round, then they calculate the same committee. One may calculate committees for rounds that the other cannot, if the blockchain is ahead, but they never calculate inconsistent committees.

This invariant is directly implied by the blockchain nonforking invariant, because (bonded and active) committees are calculated from the blockchain. Since blockchains do not fork, the shorter blockchain is a prefix of the longer one, and therefore the same committees are calculated on the common prefix. If the two blockchains are equal, they can calculate committees at exactly the same rounds.

C Background on ACL2

ACL2 [22] is a general-purpose interactive theorem prover based on an untyped first-order classical logic of total functions that is an extension of a purely functional subset of Common Lisp [42]. Predicates are functions and formulas are terms; they are false when their value is `nil`, and true when their value is `t` or anything non-`nil`.

The ACL2 syntax is consistent with Lisp. A function application is a parenthesized list consisting of the function's name followed by the arguments, e.g. $x + 2 \times f(y)$ is written `(+ x (* 2 (f y)))`. Comments extend from semicolons to line endings.

The user interacts with ACL2 by submitting a sequence of theorems, function definitions, etc. ACL2 attempts to prove theorems automatically, via algorithms similar to NQTHM [11], most notably simplification and induction. The user guides these proof attempts mainly by (i) proving lemmas for use by specific proof algorithms (e.g. rewrite rules for the simplifier) and (ii) supplying theorem-specific 'hints' (e.g. to case-split on certain conditions).

The factorial function can be defined as

```
(defun fact (n)
  (if (zp n)
      1
    (* n (fact (- n 1)))))
```

where `zp` tests if `n` is 0 or not a natural number. Thus `fact` treats arguments that are not natural numbers as 0. ACL2 functions often handle arguments of the wrong type by explicitly or implicitly coercing them to the right type—since the logic is untyped, in ACL2 a 'type' is just any subset of the universe of values. The

function `fact` is defined in the formal logic of ACL2 and thus can be reasoned about (see below); it is also a Lisp function that can be executed.

To preserve logical consistency, recursive function definitions must be proved to terminate via a measure of the arguments that decreases in each recursive call according to a well-founded relation. For `fact`, ACL2 automatically finds a measure and proves that it decreases according to a standard well-founded relation, but sometimes the user has to supply a measure.

A theorem saying that `fact` is above its argument can be introduced as

```
(defthm above
  (implies (natp n)
           (>= (fact n) n)))
```

where `natp` tests if `n` is a natural number. ACL2 proves this theorem automatically (if a standard arithmetic library is loaded), finding and using an appropriate induction rule—the one derived from the recursive definition of `fact`, in this case.

ACL2 provides logical-consistency-preserving mechanisms to axiomatize new functions, such as indefinite description functions. A function constrained to be strictly below `fact` can be introduced as

```
(defchoose below (b) (n)
  (and (natp b)
       (< b (fact n))))
```

where `b` is the variable bound by the indefinite description. This introduces the logically conservative axiom that, for every `n`, `(below n)` is a natural number less than `(fact n)`, if any exists—otherwise, `(below n)` is unconstrained. This function has a logical definition, but does not have an executable Lisp counterpart.

ACL2's Lisp-like macro mechanism provides the ability to extend the language with new constructs defined in terms of existing constructs. For instance, despite the lack of built-in quantification in the logic, functions with top-level quantifiers can be introduced. The existence of a value strictly between `fact` and `below` can be expressed by a predicate as

```
(defun-sk between (n)
  (exists (m)
          (and (natp m)
               (< (below n) m)
               (< m (fact n)))))
```

where `defun-sk` is a macro defined in terms of `defchoose` and `defun`, following a known construction [6].

A standard ACL2 library provides a macro wrapper `define` of `defun`, which features conveniences such as input and output types. For instance, the factorial function could be defined as

```
(define fact ((n natp))
  :returns (n-fact natp)
  (if (zp n)
      1
    (* n (fact (- n 1)))))
```

This is logically identical to the `defun` shown above, but the `natp` next to the parameter `n` adds an ACL2 guard, i.e. a precondition to calling the function, namely that `fact` can be only called on a natural number.[10] This requirement is enforced at every place where `fact` is called: ACL2 generates a proof obligation, i.e. attempts to prove a theorem, showing that the argument passed to `fact` is a natural number. The `:returns` in the `define` above concisely expresses a theorem saying that the result of `fact` is always a natural number; `n-fact` is used as the name of the result in the theorem.

The same ACL2 library provides a macro wrapper `define-sk` of `defun-sk`, which features similar conveniences to `define`. For example, the `between` function above could be defined as

```
(define-sk between ((n natp))
  :returns (yes/no booleanp)
  (exists (m)
          (and (natp m)))
               (< (below n) m)
               (< m (fact n)))
```

[10] ACL2's built-in `defun` also has an option to specify guards, but in a syntactically more verbose way than `define`.

D ACL2 Formalization Samples

This appendix provides some samples of the ACL2 formalization of the model presented in Section 2 and of the ACL2 invariants and proofs overviewed in Section 3 and Appendix B.

D.1 States and Events

The states and events in Figure 1 are formalized as algebraic data types, using an existing macro library to emulate such types in the untyped logic of ACL2 [43]. Each type is emulated by introducing a predicate that recognizes (i.e. returns `t` on) the values of that type, constructors and deconstructors for such values, and theorems to facilitate reasoning about values of the type (e.g. that an operation to update a component does not affect other components).

For example, the set C of certificates is formalized as

```
(fty::defprod certificate
  ((author address)
   (round pos)
   (transactions transaction-list)
   (previous address-set)
   (endorsers address-set))
  :pred certificatep)
```

which defines a record type, called `certificate`, with recognizer predicate `certificatep`,[11] and consisting of: field `author`, with the type `address` of addresses, formalizing the $a \in A$ component; field `round`, with the type `pos` of positive integers, formalizing the $r \in R$ component; field `transactions`, with the type `transaction-list` of lists of transactions, formalizing the $\overline{x} \in \overline{X}$ component; field `previous`, with the type `address-set` of sets of addresses, formalizing the $\widetilde{p} \in \widetilde{A}$ component; and field `endorsers`, with the type `address-set` of sets of addresses, formalizing the $\widetilde{q} \in \widetilde{A}$ component.

As another example, the set X of transactions is formalized as

```
(fty::deftagsum transaction
  (:bond ((validator address) (stake pos)))
  (:unbond ((validator address)))
  (:other ((unwrap any)))
  :pred transactionp)
```

which defines a variant record type, called `transaction`, with recognizer predicate `transactionp`, and consisting of: variant `:bond`, whose values consist of a validator address and a stake amount, formalizing transactions $\langle \text{BOND}, a, k \rangle \in X_{\text{B}}$; variant `:unbond`, whose values consists of a validator address, formalizing transactions $\langle \text{UNBOND}, a \rangle \in X_{\text{U}}$; and variant `:other`, whose values consist of anything, formalizing transactions in X_{O}.

The set S of system states is formalized as

```
(fty::defprod system-state
  ((validators validators-state)
   (network message-set))
  :pred system-statep)
```

where `validators-state` is the type of finite maps from addresses of type `address` to validator states of type `validator-state` (not shown here, which formalizes the set V), and `message-set` is the type of sets of messages.

The set E of events is formalized as

```
(fty::deftagsum event
  (:create ((certificate certificate)))
  (:accept ((message message)))
  (:advance ((validator address)))
  (:commit ((validator address)))
  :pred eventp)
```

[11] It is a Common Lisp and ACL2 convention to end predicate names with `p`.

which consists of a variant for each of E_{CC}, E_{CA}, E_{RA}, and E_{AC}.

The subset $I \subseteq S$ of initial states is formalized as a predicate `system-initp` over the type `system-state`. The predicate requires the `network` field to be empty, and each validator state in the `validators` field to have the value returned by the nullary function

```
(define validator-init ()
  :returns (vstate validator-statep)
  (make-validator-state :round 1
                        :dag nil
                        :endorsed nil
                        :last 0
                        :blockchain nil
                        :committed nil))
```

which uses the constructor `make-validator-state` to return a validator state of type `validator-state`, whose fields corresponds to the components of the elements of V; in ACL2, `nil` represents both the empty set and the empty list, as well as logical falsehood as described in Appendix C.

Accompanying the definitions of the types of states, events, and their constituents, we also introduce operations on these types, and prove theorems about them. For example, an operation to return all the certificates in a set that have a given round is defined as

```
(define certs-with-round ((round posp) (certs certificate-setp))
  :returns (certs-with-round certificate-setp)
  (b* (((when (set::emptyp certs)) nil)
       ((certificate cert) (set::head certs)))
    (if (equal (pos-fix round) cert.round)
        (set::insert (certificate-fix cert)
                     (certs-with-round round (set::tail certs)))
      (certs-with-round round (set::tail certs)))))
```

The `b*` is a sequential 'let' with additional conveniences, such as the early-exit `when`. If the set of certificates `certs` is empty, the operation returns the empty set `nil`. Otherwise, it picks the smallest element of the set,[12] for which there are two cases: if the round of the certificate is the given `round`, the function returns the certificate along with the ones obtained from the rest of the set;[13] otherwise, the certificate is skipped, and the rest of the set is processed. The line with `set::head` binds not only `cert` to the smallest certificate in the set, but also `cert.author`, `cert.round`, etc. to the components `(certificate->author cert)`, `(certificate->round cert)`, etc. of `cert`. The `pos-fix` and `certificate-fix` are technicalities to logically treat inputs of the wrong types as if they had the right type, since in ACL2 guards are extra-logical and functions are total.

To have ACL2 handle low-level proof details automatically, operations like `certs-with-round` must be accompanied by theorems about properties of interest, such as

```
(defthm certs-with-round-monotone
  (implies (and (certificate-setp certs1)
                (certificate-setp certs2)
                (set::subset certs1 certs2))
           (set::subset (certs-with-round round certs1)
                        (certs-with-round round certs2))))
```

This asserts the monotonicity of `certs-with-round`: if called on a subset of an input, it returns a subset of the output. The `implies` is logical implication, i.e. $\Longrightarrow$.

D.2 Auxiliary Constants, Functions, and Relations

The auxiliary constants, functions, and relations in Figure 3 and Figure 8 are formalized as ACL2 functions.

For example, the function *last* in Figure 3 is formalized by the function

[12] Finite sets in ACL2 are represented as totally ordered lists. The smallest element of a non-empty set is the first one in the list, which `set::head` returns.

[13] The `set::tail` operation returns the remaining element of a non-empty set after removing the smallest one.

```
(define blocks-last-round ((blocks block-listp))
  :returns (last natp)
  (if (consp blocks)
      (block->round (car blocks))
    0))
```

In Section 2, blockchains go from left to right (see Section 2.5), but in ACL2 we represent them in reverse, because it is easier to access and extend Lisp lists from the left. In `blocks-last-round`, `consp` tests if the list of blocks `blocks` is not empty, in which case the function returns the round of the first (i.e. accessed via `car`) block of the list; 0 is returned if the list is empty.

As another example, the part of *cmt* that updates a committee with a transaction is formalized by the function

```
(define update-committee-with-transaction ((trans transactionp) (commtt committeep))
  :returns (new-commtt committeep)
  (transaction-case
   trans
   :bond (b* ((members-with-stake (committee->members-with-stake commtt))
              (member-with-stake (omap::assoc trans.validator members-with-stake))
              (new-stake (if member-with-stake
                             (+ trans.stake (cdr member-with-stake))
                           trans.stake))
              (new-members-with-stake
               (omap::update trans.validator new-stake members-with-stake)))
           (committee new-members-with-stake))
   :unbond (b* ((members-with-stake (committee->members-with-stake commtt))
                (new-members-with-stake (omap::delete trans.validator members-with-stake)))
             (committee new-members-with-stake))
   :other (committee-fix commtt)))
```

which takes as inputs a transaction `trans` and a committee `commtt`, and returns as output a new committee `new-commtt`. This function is defined by cases on `trans`, with two subcases for `:bond` transactions, mirroring the first four defining clauses of *cmt* in Figure 3. The type `committee` of committees is a wrapper of a map from addresses to positive integers; in the function above, `members-with-stake` is that map. The map operation `omap::assoc` looks up a key in a map, returning `nil` if not found, or the key-value pair if found,[14] from which `cdr` extract the value component. Dotted variables like `trans.stake` are implicitly bound by the `transaction-case` macro, which is specific to the type `transaction` of transactions.

The calculation of the stake of a set of members of a committee is formalized by the function

```
(define committee-members-stake ((members address-setp) (commtt committeep))
  :guard (set::subset members (committee-members commtt))
  :returns (stake natp)
  (cond ((set::emptyp (address-set-fix members)) 0)
        (t (+ (committee-member-stake (address-fix (set::head members)) commtt)
              (committee-members-stake (set::tail members) commtt)))))
```

where the `:guard` extends the type requirements on the inputs by saying that the addresses must be of validators in the committee. This is defined by recursion on the set of addresses, similarly to `certs-with-round` on the set of certificates.

These and the other auxiliary functions are accompanied by theorems to help ACL2 automate the low-level details of proofs involving those functions. For example, a key property needed in quorum intersection is expressed by the theorem

```
(defthm committee-members-stake-of-union
  (implies (and (address-setp members1)
                (address-setp members2))
           (equal (committee-members-stake (set::union members1 members2) commtt)
```

[14] Returning just the value would confuse the case of an absent key with the case of a `nil` value associated to the key. This is why `omap::assoc`, which is defined on all types of maps, returns the key-value pair if the key is found.

```
                (- (+ (committee-members-stake members1 commtt)
                      (committee-members-stake members2 commtt))
                   (committee-members-stake (set::intersect members1 members2) commtt)))))
```

which relates the total stake of the union of two sets of members to the total stakes of their interesection and of the two sets.

D.3 Transitions

The transistion rules in Figure 2 are formalized by the predicate

```
(define event-possiblep ((event eventp) (systate system-statep))
  :returns (yes/no booleanp)
  (event-case
   event
   :create (create-possiblep event.certificate systate)
   :accept (accept-possiblep event.message systate)
   :advance (advance-possiblep event.validator systate)
   :commit (commit-possiblep event.validator systate)))
```

which says whether an event is possible in a state, and the function

```
(define event-next ((event eventp) (systate system-statep))
  :guard (event-possiblep event systate)
  :returns (new-systate system-statep)
  (event-case
   event
   :create (create-next event.certificate systate)
   :accept (accept-next event.message systate)
   :advance (advance-next event.validator systate)
   :commit (commit-next event.validator systate)))
```

which maps an event and a state where the event is possible to the new state after the event.

These dispatch to predicates and functions for the various kinds of events. For example, the predicate and function for round advancement are

```
(define advance-possiblep ((val addressp) (systate system-statep))
  :returns (yes/no booleanp)
  (set::in (address-fix val) (correct-addresses systate)))
```

and

```
(define advance-next ((val addressp) (systate system-statep))
  :guard (advance-possiblep val systate)
  :returns (new-systate system-statep)
  (b* (((validator-state vstate) (get-validator-state val systate))
       (new-round (1+ vstate.round))
       (new-vstate (change-validator-state vstate :round new-round))
       (new-systate (update-validator-state val new-vstate systate)))
    new-systate))
```

Comparing these definitions with the rule in Figure 2: `systate` is s; `(correct-addresses systate)` is $\mathcal{D}(\vec{v})$; `advance-possiblep` checks $a \in \mathcal{D}(\vec{v})$; `vstate` is v; `new-vstate` is v'; `new-systate` is s'; and `advance-next` increments the round number by 1.

The `...-possiblep` predicates and `...-next` functions for the other kinds of events are more complex. For examples, the function for anchor commitment is

```
(define commit-next ((val addressp) (systate system-statep))
  :guard (commit-possiblep val systate)
  :returns (new-systate system-statep)
  (b* (((validator-state vstate) (get-validator-state val systate))
```

```
      (commit-round (1- vstate.round))
      (commtt (active-committee-at-round commit-round vstate.blockchain))
      (leader (leader-at-round commit-round commtt))
      (anchor (cert-with-author+round leader commit-round vstate.dag))
      (anchors
       (collect-anchors anchor (- commit-round 2) vstate.last vstate.dag vstate.blockchain))
      ((mv new-blockchain new-committed)
       (extend-blockchain anchors vstate.dag vstate.blockchain vstate.committed))
      (new-vstate (change-validator-state vstate
                                          :last commit-round
                                          :blockchain new-blockchain
                                          :committed new-committed))
      (new-systate (update-validator-state val new-vstate systate)))
    new-systate))
```

Without getting into the details, this function calculates, in sequence: the validator's state, the round of the anchor to be committed, the committee active at that round, the leader of that committee, the anchor authored by the leader, the sequence of anchors to commit, the extended blockchain and committed certificate set, the new validtor's state, and the new system state. The guard `commit-possiblep` ensures that all these calculations are well-defined.

D.4 Invariants

Blockchain nonforking and other invariants are formalized as ACL2 predicates.

For example, the blockchain nonforking invariant in Figure 7 is formalized by the predicate

```
(define-sk nonforking-blockchains-p ((systate system-statep))
  :returns (yes/no booleanp)
  (forall (val1 val2)
          (implies (and (set::in val1 (correct-addresses systate))
                        (set::in val2 (correct-addresses systate)))
                   (lists-noforkp
                    (validator-state->blockchain (get-validator-state val1 systate))
                    (validator-state->blockchain (get-validator-state val2 systate))))))
```

which makes use of a more general function `lists-noforkp` on lists, which is also used to formalize anchor nonforking.

As another example, the DAG nonequivocation invariant in Figure 7 is formalized by the predicate

```
(define-sk unequivocal-dags-p ((systate system-statep))
  :returns (yes/no booleanp)
  (forall (val1 val2)
          (implies (and (set::in val1 (correct-addresses systate))
                        (set::in val2 (correct-addresses systate)))
                   (certificate-sets-unequivocalp
                    (validator-state->dag (get-validator-state val1 systate))
                    (validator-state->dag (get-validator-state val2 systate))))))
```

which makes use of the predicate

```
(define-sk certificate-sets-unequivocalp ((certs1 certificate-setp) (certs2 certificate-setp))
  :returns (yes/no booleanp)
  (forall (cert1 cert2)
          (implies (and (set::in cert1 certs1)
                        (set::in cert2 certs2)
                        (equal (certificate->author cert1) (certificate->author cert2))
                        (equal (certificate->round cert1) (certificate->round cert2)))
                   (equal cert1 cert2))))
```

D.5 Theorems

Some theorems have already been shown in Appendix D.1 and Appendix D.2. Every ACL2 function in the formalization is accompanied by theorems, which range in complexity. At the top-level, there are theorems showing that the invariants hold in all reachable states, under fault tolerance assumptions.

The main theorem states that blockchains never fork:

```
(defthm nonforking-blockchain-p-when-reachable
  (implies (and (system-initp systate)
                (events-possiblep events systate)
                (all-system-committees-fault-tolerant-p systate events))
           (nonforking-blockchains-p (events-next events systate))))
```

Here $s \in \widehat{H}$ is "decomposed" into its definition: `systate` is an initial state, and `events` is a list of zero or more events that take the initial state to a reachable state; `events-possiblep` and `events-next` lift `event-possiblep` and `event-next`, shown in Appendix D.3, to lists. The `all-system-committees-fault-tolerant-p` hypothesis restricts the reachable state to be fault-tolerant-reachable.

The theorems for the other invariants are stated similarly. Some of them omit the fault tolerance assumption, because unneeded.

References

[1] Aleo 2024. The Aleo Blockchain. https://aleo.org

[2] Aleo Team and Provable Team. 2019. snarkOS. https://github.com/AleoNet/snarkOS

[3] Aleo Team and Provable Team. 2020. snarkVM. https://github.com/AleoNet/snarkVM

[4] Musab A. Alturki, Jing Chen, Victor Luchangco, Brandon Moore, Karl Palmskog, Lucas Peña, and Grigore Roşu. 2020. *Towards a Verified Model of the Algorand Consensus Protocol in Coq.* Springer International Publishing, 362–367. https://doi.org/10.1007/978-3-030-54994-7_27

[5] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2024. Shoal++: High Throughput DAG BFT Can Be Fast! arXiv:2405.20488 [cs.DC] https://arxiv.org/abs/2405.20488

[6] Jeremy Avigad and Richard Zach. 2016. The Epsilon Calculus. In *The Stanford Encyclopedia of Philosophy* (summer 2016 ed.), Edward N. Zalta (Ed.). Metaphysics Research Lab, Stanford University. https://plato.stanford.edu/archives/sum2016/entries/epsilon-calculus/.

[7] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2024. Mysticeti: Reaching the Limits of Latency with Uncertified DAGs. arXiv:2310.14821v4 [cs.DC] https://arxiv.org/abs/2310.14821v4

[8] Leemon Baird. 2016. *The Swirlds hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance.* Technical Report SWIRLDS-TR-2016-01. Swirlds Inc. Revision date: March 18, 2018.

[9] Nathalie Bertrand, Pranav Ghorpade, Sasha Rubin, Bernhard Scholz, and Pavle Subotic. 2024. Reusable Formal Verification of DAG-based Consensus Protocols. arXiv:2407.02167v1 [cs.LO] https://arxiv.org/abs/2407.02167v1

[10] Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazić, Pierre Tholoniat, and Josef Widder. 2022. Holistic Verification of Blockchain Consensus. arXiv:2206.04489 [cs.CR] https://arxiv.org/abs/2206.04489

[11] Robert S. Boyer and J Strother Moore. 1979. *The Boyer-Moore Theorem Prover.* Academic Press. https://www.cs.utexas.edu/users/boyer/ftp/nqthm/.

[12] Harold Carr, Christopher Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. 2022. Towards Formal Verification of HotStuff-based Byzantine Fault Tolerant Consensus in Agda: Extended Version. arXiv:2203.14711v2 [cs.DC] https://arxiv.org/abs/2203.14711v2

[13] Sang-Min Choi, Jiho Park, Quan Nguyen, Andre Cronje, Kiyoung Jang, Hyunjoon Cheon, Yo-Sub Han, and Byung-Ik Ahn. 2018. OPERA: Reasoning about continuous common knowledge in asynchronous distributed systems. arXiv:1810.02186 [cs.DC] https://arxiv.org/abs/1810.02186

[14] Alessandro Coglio and Eric McCarthy. [n. d.]. ACL2 Formal Model and Proofs of AleoBFT. https://github.com/acl2/acl2/tree/master/books/projects/aleo/bft

[15] Karl Crary. 2021. Verifying the Hashgraph Consensus Algorithm. arXiv:2102.01167 [cs.LO] https://arxiv.org/abs/2102.01167

[16] Xiaohai Dai, Guanxiong Wang, Jiang Xiao, Zhengxuan Guo, Rui Hao, Xia Xie, and Hai Jin. 2024. LightDAG: A Low-latency DAG-based BFT Consensus through Lightweight Broadcast. Cryptology ePrint Archive, Paper 2024/160. https://eprint.iacr.org/2024/160

[17] Xiaohai Dai, Zhaonan Zhang, Jiang Xiao, Jingtao Yue, Xia Xie, and Hai Jin. 2024. GradedDAG: An Asynchronous DAG-based BFT Consensus with Lower Latency. Cryptology ePrint Archive, Paper 2024/142. https://eprint.iacr.org/2024/142

[18] George Danezis and David Hrycyszyn. 2018. Blockmania: from Block DAGs to Consensus. arXiv:1809.01620v2 [cs.CR] https://arxiv.org/abs/1809.01620v2

[19] George Danezis, Eleftherios Kokoris Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. arXiv:2105.11827v4 [cs.CR] https://arxiv.org/abs/2105.11827v4

[20] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał undefinedwiundefinedtek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies* (Zurich, Switzerland) *(AFT '19)*. Association for Computing Machinery, New York, NY, USA, 214–228. https://doi.org/10.1145/3318041.3355467

[21] Philipp Jovanovic, Lefteris Kokoris Kogias, Bryan Kumara, Alberto Sonnino, Pasindu Tennage, and Igor Zablotchi. 2024. Mahi-Mahi: Low-Latency Asynchronous BFT DAG-Based Consensus. arXiv:2410.08670v2 [cs.DC] https://arxiv.org/abs/2410.08670v2

[22] Matt Kaufmann and J Strother Moore. 1990. The ACL2 Theorem Prover: Web Site. http://acl2.org

[23] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need is DAG. arXiv:2102.08325 [cs.DC] https://arxiv.org/abs/2102.08325

[24] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. 2023. Cordial Miners: Fast and Efficient Consensus for Every Eventuality. In *37th International Symposium on Distributed Computing (DISC 2023)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.DISC.2023.26

[25] Yanhong A. Liu, Saksham Chand, and Scott D. Stoller. 2019. Moderately Complex Paxos Made Simple: High-Level Executable Specification of Distributed Algorithms. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP '19)*. ACM, 1–15. https://doi.org/10.1145/3354166.3354180

[26] Yanhong A. Liu and Scott D. Stoller. 2024. Tutorial: Consensus Algorithms from Classical to Blockchain: Quickly Program, Configure, Run, and Check. In *2024 IEEE 44th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. 1–4. https://doi.org/10.1109/ICDCSW63686.2024.00005

[27] Giuliano Losa. 2021. *Formally Verifying the Tendermint Blockchain Protocol*. Technical Report. Galois. https://galois.com/blog/2021/07/formally-verifying-the-tendermint-blockchain-protocol/ Accessed: 2024-11-01.

[28] Giuliano Losa and Mike Dodds. 2020. On the Formal Verification of the Stellar Consensus Protocol. In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020) (Open Access Series in Informatics (OASIcs), Vol. 84)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:9. https://doi.org/10.4230/OASIcs.FMBC.2020.9

[29] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2024. BBCA-CHAIN: Low Latency, High Throughput BFT Consensus on a DAG. arXiv:2310.06335 [cs.DC] https://arxiv.org/abs/2310.06335

[30] Dahlia Malkhi and Pawel Szalachowski. 2022. Maximal Extractable Value (MEV) Protection on a DAG. arXiv:2208.00940v4 [cs.CR] https://arxiv.org/abs/2208.00940v4

[31] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System.

[32] Quan Nguyen, Andre Cronje, Michael Kong, Egor Lysenko, and Alex Guzev. 2021. Lachesis: Scalable Asynchronous BFT on DAG Streams. arXiv:2108.01900 [cs.DC] https://arxiv.org/abs/2108.01900

[33] M. Praveen, Raghavendra Ramesh, and Isaac Doidge. 2024. Formally Verifying the Safety of Pipelined Moonshot Consensus Protocol. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024) (Open Access Series in Informatics (OASIcs), Vol. 118)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:16. https://doi.org/10.4230/OASIcs.FMBC.2024.3

[34] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *European Symposium on Programming*. https://api.semanticscholar.org/CorpusID:4899635

[35] Sean Rowan and Na iri Usher. 2019. Flare Consensus Protocol. https://flare.network/wp-content/uploads/FCP-White-Paper.pdf

[36] Maria A. Schett and George Danezis. 2021. Embedding a Deterministic BFT Protocol in a Block DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 177–186. https://doi.org/10.1145/3465084.3467930

[37] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2024. Sailfish: Towards Improving the Latency of DAG-based BFT. Cryptology ePrint Archive, Paper 2024/472. https://eprint.iacr.org/2024/472

[38] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. 2023. Shoal: Improving DAG-BFT Latency And Robustness. arXiv:2306.03058v2 [cs.DC] https://arxiv.org/abs/2306.03058v2

[39] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. arXiv:2201.05677v3 [cs.CR] https://arxiv.org/abs/2201.05677v3

[40] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: The Partially Synchronous Version. arXiv:2209.05633v1 [cs.DC] https://arxiv.org/abs/2209.05633v1

[41] Chrysoula Stathakopoulou, Michael Wei, Maofan Yin, Hongbo Zhang, and Dahlia Malkhi. 2023. BBCA-LEDGER: High Throughput Consensus meets Low Latency. arXiv:2306.14757 [cs.DC] https://arxiv.org/abs/2306.14757

[42] Guy L. Steele. 1984. *Common Lisp the Language*. Digital Press.

[43] Sol Swords and Jared Davis. 2015. Fix Your Types. In *Proc. 13th International Workshop on the ACL2 Theorem Prover and Its Applications.*

[44] Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz, Seth Gilbert, and Ilya Sergey. 2024. Compositional Verification of Composite Byzantine Protocols. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 34–48. https://doi.org/10.1145/3658644.3690355

[45] zkSecurity. 2023. Audit of Aleo's consensus. https://www.zksecurity.xyz/reports/aleo-consensus