A Formal Specification of the Cardano Consensus

Deliverable XXXX

Javier Díaz <javier.diaz@iohk.io>

Project: Cardano

Type: Deliverable
Due Date: XXXX

Responsible team: Formal Methods Team
Editor: Javier Díaz, IOHK
Team Leader: James Chapman, IOHK

Version 1.0
XXXX

Dissemination Level		
PU	Public	√
CO	Confidential, only for company distribution	
DR	Draft, not for general circulation	

Contents

List of Figures

Change Log

Rev.	Date	Who	Team	What
1	2024/03/01	Javier Díaz	FM (IOHK)	Initial version (0.1).

A Formal Specification of the Cardano Consensus

Javier Díaz
javier.diaz@iohk.io

December 10, 2024

Abstract

This document provides a formal specification of the Cardano consensus layer.

List of Contributors

...

1 Transition Rule Dependencies

Figure 1 shows all STS rules, the sub-rules they use and possible dependencies. Each node in the graph represents one rule, the top rule being CHAINHEAD. A straight arrow from one node to another one represents a sub-rule relationship.

An arrow with a dotted line from one node to another represents a dependency in the sense that the output of the target rule is an input to the source one, either as part of the source state, the environment or the signal. These dependencies are between sub-rules of a rule.

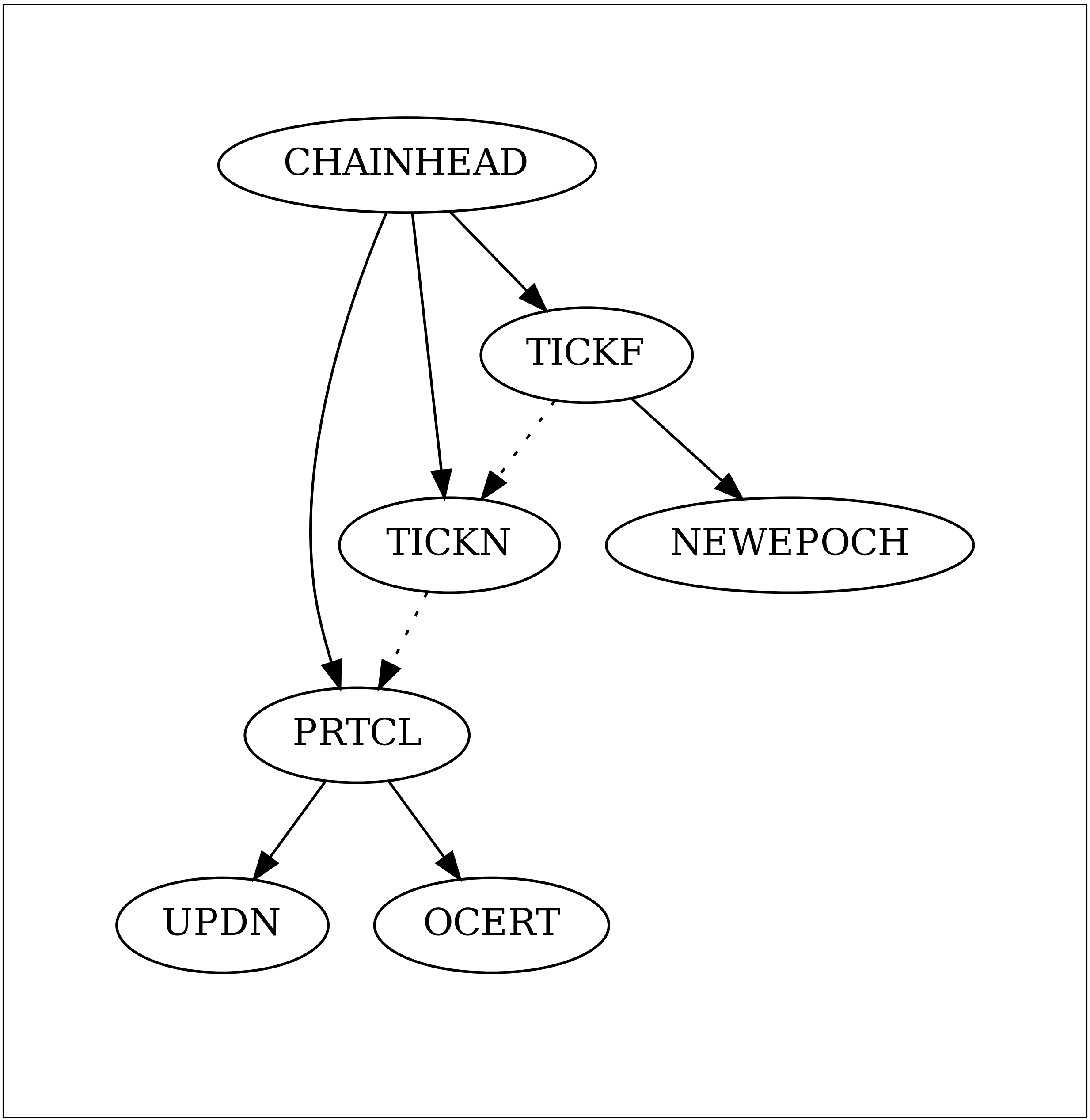

Figure 1: STS Rules, Sub-Rules and Dependencies

2 Common Interface

2.1 NEWEPOCH Transition

The New Epoch Transition (NEWEPOCH) performs some upkeep when a new epoch begins.

New Epoch Transitions

$$\vdash \_ \xrightarrow[\mathrm{NEWEPOCH}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{NewEpochState} \times \mathsf{Epoch} \times \mathsf{NewEpochState})$$

Figure 2: New Epoch transition-system types

3 Blockchain Layer

This chapter introduces the view of the blockchain layer as required for the consensus. The main transition rule is CHAINHEAD which calls the subrules TICKF, TICKN and PRTCL.

3.1 Block Definitions

Abstract types

$$
\begin{array}{rlr}
h \in & \mathsf{HashHeader} & \text{hash of a block header} \\
hb \in & \mathsf{HashBBody} & \text{hash of a block body} \\
bn \in & \mathsf{BlockNo} & \text{block number} \\
\eta \in & \mathsf{Nonce} & \text{nonce} \\
vrfRes \in & \mathsf{VRFRes} & \text{VRF result value}
\end{array}
$$

Operational Certificate

$$
\mathsf{OCert} = \left(
\begin{array}{rlr}
vk_{hot} \in & \mathsf{VKey_{ev}} & \text{operational (hot) key} \\
n \in & \mathbb{N} & \text{certificate issue number} \\
c_0 \in & \mathsf{KESPeriod} & \text{start KES period} \\
\sigma \in & \mathsf{Sig} & \text{cold key signature}
\end{array}
\right)
$$

Block Header Body

$$
\mathsf{BHBody} = \left(
\begin{array}{rlr}
prevHeader \in & \mathsf{HashHeader}^{?} & \text{hash of previous block header} \\
issuerVk \in & \mathsf{VKey} & \text{block issuer} \\
vrfVk \in & \mathsf{VKey} & \text{VRF verification key} \\
blockNo \in & \mathsf{BlockNo} & \text{block number} \\
slot \in & \mathsf{Slot} & \text{block slot} \\
vrfRes \in & \mathsf{VRFRes} & \text{VRF result value} \\
vrfPrf \in & \mathsf{Proof} & \text{VRF proof} \\
bodySize \in & \mathbb{N} & \text{size of the block body} \\
bodyHash \in & \mathsf{HashBBody} & \text{block body hash} \\
oc \in & \mathsf{OCert} & \text{operational certificate} \\
pv \in & \mathsf{ProtVer} & \text{protocol version}
\end{array}
\right)
$$

Block Types

$$
bh \in \mathsf{BHeader} \quad = \quad \mathsf{BHBody} \times \mathsf{Sig}
$$

Abstract functions

$$
\begin{array}{rlr}
\star & \in \mathsf{Nonce} \to \mathsf{Nonce} \to \mathsf{Nonce} & \text{binary nonce operation} \\
\mathsf{headerHash} & \in \mathsf{BHeader} \to \mathsf{HashHeader} & \text{hash of a block header} \\
\mathsf{headerSize} & \in \mathsf{BHeader} \to \mathbb{N} & \text{size of a block header} \\
\mathsf{slotToSeed} & \in \mathsf{Slot} \to \mathsf{Seed} & \text{convert a slot to a seed} \\
\mathsf{nonceToSeed} & \in \mathsf{Nonce} \to \mathsf{Seed} & \text{convert a nonce to a seed} \\
\mathsf{prevHashToNonce} & \in \mathsf{HashHeader}^{?} \to \mathsf{Seed} & \text{convert an optional header hash to a seed}
\end{array}
$$

Accessor Functions

$$
\begin{array}{rlrl}
\mathsf{blockHeader} \in & \mathsf{Block} \to \mathsf{BHeader} & \mathsf{headerBody} \in & \mathsf{BHeader} \to \mathsf{BHBody} \\
\mathsf{headerSig} \in & \mathsf{BHeader} \to \mathsf{Sig} & \mathsf{hBVkCold} \in & \mathsf{BHBody} \to \mathsf{VKey} \\
\mathsf{hBVrfVk} \in & \mathsf{BHBody} \to \mathsf{VKey} & \mathsf{hBPrevHeader} \in & \mathsf{BHBody} \to \mathsf{HashHeader}^{?} \\
\mathsf{hBSlot} \in & \mathsf{BHBody} \to \mathsf{Slot} & \mathsf{hBBlockNo} \in & \mathsf{BHBody} \to \mathsf{BlockNo} \\
\mathsf{hBVrfRes} \in & \mathsf{BHBody} \to \mathsf{VRFRes} & \mathsf{hBVrfPrf} \in & \mathsf{BHBody} \to \mathsf{Proof} \\
\mathsf{hBBodySize} \in & \mathsf{BHBody} \to \mathbb{N} & \mathsf{hBBodyHash} \in & \mathsf{BHBody} \to \mathsf{HashBBody} \\
\mathsf{hBOCert} \in & \mathsf{BHBody} \to \mathsf{OCert} & \mathsf{hBProtVer} \in & \mathsf{BHBody} \to \mathsf{ProtVer}
\end{array}
$$

Figure 3: Block Definitions

Block Helper Functions

$$\mathsf{hBLeader} \in \mathsf{BHBody} \to \mathbb{N}$$
$$\mathsf{hBLeader}\ bhb = \mathsf{hash}\ (\text{“}L\text{”} \mid (\mathsf{hBVrfRes}\ bhb))$$

$$\mathsf{hBNonce} \in \mathsf{BHBody} \to \mathsf{Nonce}$$
$$\mathsf{hBNonce}\ bhb = \mathsf{hash}\ (\text{“}N\text{”} \mid (\mathsf{hBVrfRes}\ bhb))$$

Figure 4: Helper Functions used for Blocks

Tick Nonce environments

$$\mathsf{TickNonceEnv} = \left(\begin{array}{rlr} \eta_c & \in \mathsf{Nonce} & \text{candidate nonce} \\ \eta_{ph} & \in \mathsf{Nonce} & \text{previous header hash as nonce} \end{array}\right)$$

Tick Nonce states

$$\mathsf{TickNonceState} = \left(\begin{array}{rlr} \eta_0 & \in \mathsf{Nonce} & \text{epoch nonce} \\ \eta_h & \in \mathsf{Nonce} & \text{nonce from hash of previous epoch's last block header} \end{array}\right)$$

Figure 5: Tick Nonce types

3.2 TICKN Transition

The Tick Nonce Transition (TICKN) is responsible for updating the epoch nonce and the previous epoch's hash nonce at the start of an epoch. Its environment is shown in Figure 5 and consists of the candidate nonce η_c and the previous epoch's last block header hash as a nonce η_{ph}. Its state consists of the epoch nonce η_0 and the previous epoch's last block header hash nonce η_h.

The signal to the transition rule TICKN is a marker indicating whether we are in a new epoch. If we are in a new epoch, we update the epoch nonce and the previous hash. Otherwise, we do nothing.

$$\text{Not-New-Epoch}\ \frac{}{\begin{array}{c}\eta_c \\ \eta_{ph}\end{array} \vdash \left(\begin{array}{c}\eta_0 \\ \eta_h\end{array}\right) \xrightarrow[\text{TICKN}]{\mathsf{False}} \left(\begin{array}{c}\eta_0 \\ \eta_h\end{array}\right)} \quad (1)$$

$$\text{New-Epoch}\ \frac{}{\begin{array}{c}\eta_c \\ \eta_{ph}\end{array} \vdash \left(\begin{array}{c}\eta_0 \\ \eta_h\end{array}\right) \xrightarrow[\text{TICKN}]{\mathsf{True}} \left(\begin{array}{c}\eta_c \star \eta_h \\ \eta_{ph}\end{array}\right)} \quad (2)$$

Figure 6: Tick Nonce rules

3.3 UPDN Transition

The Update Nonce Transition (UPDN) updates the nonces until the randomness gets fixed. The environment is shown in Figure 7 and consists of the block nonce η. The state is shown in Figure 7 and consists of the candidate nonce η_c and the evolving nonce η_v.

The transition rule UPDN takes the slot s as signal. There are two different cases for UPDN: one where s is not yet RandomnessStabilisationWindow[1] slots from the beginning of the next epoch and one where s is less than RandomnessStabilisationWindow slots until the start of the next epoch.

Note that in 3, the candidate nonce η_c transitions to $\eta_v \star \eta$, not $\eta_c \star \eta$. The reason for this is that even though the candidate nonce is frozen sometime during the epoch, we want the two nonces to again be equal at the start of a new epoch.

3.4 OCERT Transition

The Operational Certificate Transition (OCERT) environment consists of the set of stake pools *stpools*. Its state is the mapping of operation certificate issue numbers. Its signal is a block header.

[1] Note that in pre-Conway eras StabilityWindow was used instead of RandomnessStabilisationWindow.

Update Nonce environments

$$\mathsf{UpdateNonceEnv} = \left(\eta \in \mathsf{Nonce} \quad \text{new nonce} \right)$$

Update Nonce states

$$\mathsf{UpdateNonceState} = \left(\begin{array}{ll} \eta_v \in \mathsf{Nonce} & \text{evolving nonce} \\ \eta_c \in \mathsf{Nonce} & \text{candidate nonce} \end{array} \right)$$

Update Nonce Transitions

$$\_ \vdash \_ \xrightarrow[\mathrm{UPDN}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{UpdateNonceEnv} \times \mathsf{UpdateNonceState} \times \mathsf{Slot} \times \mathsf{UpdateNonceState})$$

Figure 7: Update Nonce transition-system types

$$\text{Update-Both}\ \frac{s < \mathsf{firstSlot}\ ((\mathsf{epoch}\ s) + 1) - \mathsf{RandomnessStabilisationWindow}}{\eta \vdash \left(\begin{array}{c} \eta_v \\ \eta_c \end{array}\right) \xrightarrow[\mathrm{UPDN}]{s} \left(\begin{array}{c} \eta_v \star \eta \\ \eta_v \star \eta \end{array}\right)} \quad (3)$$

$$\text{Only-Evolve}\ \frac{s \geq \mathsf{firstSlot}\ ((\mathsf{epoch}\ s) + 1) - \mathsf{RandomnessStabilisationWindow}}{\eta \vdash \left(\begin{array}{c} \eta_v \\ \eta_c \end{array}\right) \xrightarrow[\mathrm{UPDN}]{s} \left(\begin{array}{c} \eta_v \star \eta \\ \eta_c \end{array}\right)} \quad (4)$$

Figure 8: Update Nonce rules

The transition rule OCERT is shown in Figure 10. From the block header body *bhb* we first extract the following:

- The operational certificate, consisting of the hot key vk_{hot}, the certificate issue number n, the KES period start c_0 and the cold key signature τ.
- The cold key vk_{cold}.
- The slot s for the block.
- The number of KES periods that have elapsed since the start period on the certificate.

Using this we verify the preconditions of the operational certificate state transition which are the following:

- The KES period of the slot in the block header body must be greater than or equal to the start value c_0 listed in the operational certificate, and less than MaxKESEvo-many KES periods after c_0. The value of MaxKESEvo is the agreed-upon lifetime of an operational certificate, see [SL-D1].
- *hk* exists as key in the mapping of certificate issues numbers to a KES period m and that period is less than or equal to n. Also, n must be less than or equal to the successor of m.
- The signature τ can be verified with the cold verification key vk_{cold}.
- The KES signature σ can be verified with the hot verification key vk_{hot}.

Operational Certificate Transitions

$$\_ \vdash \_ \xrightarrow[\text{OCERT}]{\_} \_ \subseteq \mathbb{P}\ (\mathbb{P}\ \mathsf{KeyHash} \times \mathsf{KeyHash}_{pool} \mapsto \mathbb{N} \times \mathsf{BHeader} \times \mathsf{KeyHash}_{pool} \mapsto \mathbb{N})$$

Operational Certificate helper function

$$\mathsf{currentIssueNo} \in \mathbb{P}\ \mathsf{KeyHash} \to (\mathsf{KeyHash}_{pool} \mapsto \mathbb{N}) \to \mathsf{KeyHash}_{pool} \to \mathbb{N}^?$$

$$\mathsf{currentIssueNo}\ stpools\ cs\ hk = \begin{cases} hk \mapsto n \in cs & n \\ hk \in stpools & 0 \\ \text{otherwise} & \diamond\ (7) \end{cases}$$

Figure 9: Operational Certificate transition-system types

After this, the transition system updates the operational certificate state by updating the mapping of operational certificates where it overwrites the entry of the key hk with the KES period n.

$$\begin{array}{ccc} (bhb, \sigma) := bh & (vk_{hot}, n, c_0, \tau) := \mathsf{hBOCert}\ bhb & vk_{cold} := \mathsf{hBVkCold}\ bhb \\ hk := \mathsf{hashKey}\ vk_{cold} & s := \mathsf{hBSlot}\ bhb & t := \mathsf{kesPeriod}\ s - c_0 \end{array}$$

$$\text{OCert}\ \dfrac{\begin{array}{c} c_0 \overset{(1)}{\leq} \mathsf{kesPeriod}\ s \overset{(2)}{<} c_0 + \mathsf{MaxKESEvo} \\ \mathsf{currentIssueNo}\ stpools\ cs\ hk = m \qquad n \in \{\overset{(3)}{m}, \overset{(4)}{m+1}\} \\ \mathcal{V}_{vk_{cold}}[\![(vk_{hot}, n, c_0)]\!]_{\tau}\ (5) \qquad \mathcal{V}^{\mathsf{KES}}_{vk_{hot}}[\![bhb]\!]^{t}_{\sigma}\ (6) \end{array}}{stpools \vdash cs \xrightarrow[\text{OCERT}]{bh} cs \underrightarrow{\cup} \{hk \mapsto n\}} \quad (5)$$

Figure 10: Operational Certificate rules

The OCERT rule has the following predicate failures:

1. If the KES period is less than the KES period start in the certificate, there is a *KESBeforeStart* failure.
2. If the KES period is greater than or equal to the KES period end (start + MaxKESEvo) in the certificate, there is a *KESAfterEnd* failure.
3. If the period counter in the original key hash counter mapping is larger than the period number in the certificate, there is a *CounterTooSmall* failure.
4. If the period number in the certificate is larger than the successor of the period counter in the original key hash counter mapping, there is a *CounterOverIncremented* failure.
5. If the signature of the hot key, KES period number and period start is incorrect, there is an *InvalidSignature* failure.
6. If the KES signature using the hot key of the block header body is incorrect, there is an *InvalideKesSignature* failure.
7. If there is no entry in the key hash to counter mapping for the cold key, there is a *NoCounterForKeyHash* failure.

$$\mathsf{issuerIDfromBHBody} \in \mathsf{BHBody} \to \mathsf{KeyHash}_{pool}$$
$$\mathsf{issuerIDfromBHBody} = \mathsf{hashKey} \circ \mathsf{hBVkCold}$$

$$\mathsf{vrfChecks} \in \mathsf{Nonce} \to \mathsf{BHBody} \to \mathsf{Bool}$$
$$\mathsf{vrfChecks}\ \eta_0\ bhb = \mathsf{verifyVrf}_{\mathsf{VRFRes}}\ vrfK\ ((\mathsf{slotToSeed}\ slot)\ \mathsf{XOR}\ (\mathsf{nonceToSeed}\ \eta_0))\ (proof,\ value)$$
where
$$slot := \mathsf{hBSlot}\ bhb$$
$$vrfK := \mathsf{hBVrfVk}\ bhb$$
$$value := \mathsf{hBVrfRes}\ bhb$$
$$proof := \mathsf{hBVrfPrf}\ bhb$$

$$\mathsf{praosVrfChecks} \in \mathsf{Nonce} \to \mathsf{PoolDistr} \to (0,\ 1] \to \mathsf{BHBody} \to \mathsf{Bool}$$
$$\mathsf{praosVrfChecks}\ \eta_0\ pd\ f\ bhb =$$
$$hk \mapsto (\sigma,\ vrfHK) \in pd\ (1)$$
$$\wedge\ vrfHK = \mathsf{hashKey}\ vrfK\ (2)$$
$$\wedge\ \mathsf{vrfChecks}\ \eta_0\ bhb\ (3)$$
$$\wedge\ \mathsf{checkLeaderVal}\ (\mathsf{hBLeader}\ bhb)\ \sigma\ f\ (4)$$
where
$$hk := \mathsf{issuerIDfromBHBody}\ bhb$$
$$vrfK := \mathsf{hBVrfVk}\ bhb$$

Figure 11: VRF helper functions

3.5 Verifiable Random Function

In this section we define a function praosVrfChecks which performs all the VRF related checks on a given block header body. In addition to the block header body, the function requires the epoch nonce, the stake distribution (aggregated by pool), and the active slots coefficient from the protocol parameters. The function checks:

- The validity of the proofs for the leader value and the new nonce.
- The verification key is associated with relative stake σ in the stake distribution.
- The hBLeader value of *bhb* indicates a possible leader for this slot. The function checkLeaderVal is defined in 5.

The definition of praosVrfChecks is shown in Figure 11 and has the following predicate failures:

1. If the VRF key is not in the pool distribution, there is a *VRFKeyUnknown* failure.
2. If the VRF key hash does not match the one listed in the block header, there is a *VRFKeyWrongVRFKey* failure.

3. If the VRF generated value in the block header does not validate against the VRF certificate, there is a *VRFKeyBadProof* failure.

4. If the VRF generated leader value in the block header is too large compared to the relative stake of the pool, there is a *VRFLeaderValueTooBig* failure.

Protocol environments

$$\mathsf{PrtclEnv} = \left(\begin{array}{rlr} pd & \in \mathsf{PoolDistr} & \text{pool stake distribution} \\ \eta_0 & \in \mathsf{Nonce} & \text{epoch nonce} \end{array}\right)$$

Protocol states

$$\mathsf{PrtclState} = \left(\begin{array}{rlr} cs & \in \mathsf{KeyHash}_{pool} \mapsto \mathbb{N} & \text{operational certificate issues numbers} \\ \eta_v & \in \mathsf{Nonce} & \text{evolving nonce} \\ \eta_c & \in \mathsf{Nonce} & \text{candidate nonce} \end{array}\right)$$

Protocol Transitions

$$\_ \vdash \_ \xrightarrow[\mathsf{PRTCL}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{PrtclEnv} \times \mathsf{PrtclState} \times \mathsf{BHeader} \times \mathsf{PrtclState})$$

Figure 12: Protocol transition-system types

3.6 PRTCL **Transition**

The Protocol Transition (PRTCL) calls the transition UPDN to update the evolving and candidate nonces, and checks the operational certificate with OCERT.

The environments for this transition are:

- The stake pool stake distribution pd.
- The epoch nonce η_0.

The states for this transition consists of:

- The operational certificate issue number mapping.
- The evolving nonce.
- The candidate nonce for the next epoch.

$$bhb := \mathsf{headerBody}\ bh \qquad \eta := \mathsf{hBNonce}\ bhb \qquad slot := \mathsf{hBSlot}\ bhb$$

$$\frac{\begin{array}{c} \eta \vdash \begin{pmatrix} \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\mathsf{UPDN}]{slot} \begin{pmatrix} \eta_v' \\ \eta_c' \end{pmatrix} \\ \mathrm{dom}\, pd \vdash cs \xrightarrow[\mathsf{OCERT}]{bh} cs' \\ \mathsf{praosVrfChecks}\ \eta_0\ pd\ \mathsf{ActiveSlotCoeff}\ bhb \end{array}}{\begin{array}{c} pd \\ \eta_0 \end{array} \vdash \begin{pmatrix} cs \\ \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\mathsf{PRTCL}]{bh} \begin{pmatrix} cs' \\ \eta_v' \\ \eta_c' \end{pmatrix}}\ \mathsf{PRTCL} \tag{6}$$

Figure 13: Protocol rules

This transition establishes that a block producer is in fact authorized. Since there are three key pairs involved (cold keys, VRF keys, and hot KES keys) it is worth examining the interaction closely.

- First we check the operational certificate with OCERT. This uses the cold verification key given in the block header. We do not yet trust that this key is a registered pool key. If this transition is successful, we know that the cold key in the block header has authorized the block.

- Next, in the vrfChecks predicate, we check that the hash of this cold key is in the mapping *pd*, and that it maps to $(\sigma,\ hk_{vrf})$, where $(\sigma,\ hk_{vrf})$ is the hash of the VRF key in the header. If praosVrfChecks returns true, then we know that the cold key in the block header was a registered stake pool at the beginning of the previous epoch, and that it is indeed registered with the VRF key listed in the header.
- Finally, we use the VRF verification key in the header, along with the VRF proofs in the header, to check that the operator is allowed to produce the block.

Tick Forecast Transitions

$$\vdash \_ \xrightarrow[\text{TICKF}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{NewEpochState} \times \mathsf{Slot} \times \mathsf{NewEpochState})$$

Tick Forecast helper function

$$\mathsf{adoptGenesisDelegs} \in \mathsf{EpochState} \to \mathsf{Slot} \to \mathsf{EpochState}$$
$$\mathsf{adoptGenesisDelegs}\ es\ slot = es'$$

where

$$(acnt,\ ss,\ (us,\ (ds,ps)),\ prevPp,\ pp) := es$$
$$(\ rewards,\ delegations,\ ptrs,\ fGenDelegs,\ genDelegs,\ i_{rwd}) := ds$$
$$curr := \{(s,\ gkh) \mapsto (vkh,\ vrf) \in fGenDelegs \mid s \leq slot\}$$
$$fGenDelegs' := fGenDelegs \setminus curr$$
$$genDelegs' := \left\{ gkh \mapsto (vkh,\ vrf) \;\middle|\; \begin{array}{l} (s,\ gkh) \mapsto (vkh,\ vrf) \in curr \\ s = \max\{s' \mid (s',\ gkh) \in \mathrm{dom}\, curr\} \end{array} \right\}$$
$$ds' := (rewards,\ delegations,\ ptrs,\ fGenDelegs',\ genDelegs \underrightarrow{\cup} genDelegs',\ i_{rwd})$$
$$es' := (acnt,\ ss,\ (us,\ (ds',ps)),\ prevPp,\ pp)$$

Figure 14: Tick Forecast transition-system types

3.7 TICKF **Transition**

The Tick Forecast Transition (TICKF) performs some chain level upkeep. The state is the epoch specific state necessary for the NEWEPOCH transition.

Part of the upkeep is updating the genesis key delegation mapping according to the future delegation mapping. For each genesis key, we adopt the most recent delegation in *fGenDelegs* that is past the current slot, and any future genesis key delegations past the current slot is removed. The helper function adoptGenesisDelegs accomplishes the update.

The TICKF transition rule is shown in Figure 15. The signal is a slot *s*.

One sub-transition is done: The NEWEPOCH transition performs any state change needed if it is the first block of a new epoch.

$$\text{TickForecast}\ \frac{\begin{array}{c} \vdash nes \xrightarrow[\text{NEWEPOCH}]{\mathsf{epoch}\ s} nes' \\ (e'_\ell,\ b'_{prev},\ b'_{cur},\ es',\ ru',\ pd') := nes' \\ es'' := \mathsf{adoptGenesisDelegs}\ es'\ s \\ forecast := (e'_\ell,\ b'_{prev},\ b'_{cur},\ es'',\ ru',\ pd') \end{array}}{\vdash nes \xrightarrow[\text{TICKF}]{s} \mathbf{forecast}} \quad (7)$$

Figure 15: Tick Forecast rules

3.8 CHAINHEAD Transition

The Chain Head Transition rule (CHAINHEAD) is the main rule of the blockchain layer part of the STS. It calls TICKF, TICKN, and PRTCL, as sub-rules.

The transition checks the following things: (via chainChecks and prtlSeqChecks from Figure 17):

- The slot in the block header body is larger than the last slot recorded.
- The block number increases by exactly one.
- The previous hash listed in the block header matches the previous block header hash which was recorded.
- The size of the block header is less than or equal to the maximal size that the protocol parameters allow for block headers.
- The size of the block body, as claimed by the block header, is less than or equal to the maximal size that the protocol parameters allow for block bodies.
- The node is not obsolete, meaning that the major component of the protocol version in the protocol parameters is not bigger than the constant MaxMajorPV.

The CHAINHEAD state is shown in Figure 16, it consists of the following:

- The operational certificate issue number map *cs*.
- The epoch nonce η_0.
- The evolving nonce η_v.
- The candidate nonce η_c.
- The previous epoch hash nonce η_h.
- The last header hash h.
- The last slot s_ℓ.
- The last block number b_ℓ.

The CHAINHEAD transition rule is shown in Figure 18. It contains a new epoch state *nes* in the environment and its signal is a block header *bh*. The transition uses a few helper functions defined in Figure 17.

The CHAINHEAD transition rule has the following predicate failures:

1. If the slot of the block header body is not larger than the last slot, there is a *WrongSlotInterval* failure.
2. If the block number does not increase by exactly one, there is a *WrongBlockNo* failure.
3. If the hash of the previous header of the block header body is not equal to the last header hash, there is a *WrongBlockSequence* failure.
4. If the size of the block header is larger than the maximally allowed size, there is a *HeaderSizeTooLarge* failure.
5. If the size of the block body is larger than the maximally allowed size, there is a *BlockSizeTooLarge* failure.
6. If the major component of the protocol version is larger than MaxMajorPV, there is a *ObsoleteNode* failure.

Chain Head states

$$\mathsf{LastAppliedBlock} = \left(\begin{array}{rlr} b_\ell \in & \mathsf{BlockNo} & \text{last block number} \\ s_\ell \in & \mathsf{Slot} & \text{last slot} \\ h \in & \mathsf{HashHeader} & \text{latest header hash} \end{array}\right)$$

$$\mathsf{ChainHeadState} = \left(\begin{array}{rlr} cs \in & \mathsf{KeyHash}_{pool} \mapsto \mathbb{N} & \text{operational certificate issue numbers} \\ \eta_0 \in & \mathsf{Nonce} & \text{epoch nonce} \\ \eta_v \in & \mathsf{Nonce} & \text{evolving nonce} \\ \eta_c \in & \mathsf{Nonce} & \text{candidate nonce} \\ \eta_h \in & \mathsf{Nonce} & \text{nonce from hash of last epoch's last header} \\ lab \in & \mathsf{LastAppliedBlock}^? & \text{latest applied block} \end{array}\right)$$

Chain Head Transitions

$$\_ \vdash \_ \xrightarrow[\mathsf{CHAINHEAD}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{NewEpochState} \times \mathsf{ChainHeadState} \times \mathsf{BHeader} \times \mathsf{ChainHeadState})$$

Figure 16: Chain Head transition-system types

Chain Head Transition Helper Functions

$$\mathsf{chainChecks} \in \mathbb{N} \to (\mathbb{N}, \mathbb{N}, \mathsf{ProtVer}) \to \mathsf{BHeader} \to \mathsf{Bool}$$

$$\begin{aligned} &\mathsf{chainChecks}\ maxpv\ (maxBHSize,\ maxBBSize,\ protocolVersion)\ bh = \\ &\quad m \overset{(6)}{\leq} maxpv \\ &\quad \wedge\ \mathsf{headerSize}\ bh \overset{(4)}{\leq} maxBHSize \\ &\quad \wedge\ \mathsf{hBBodySize}\ (\mathsf{headerBody}\ bh) \overset{(5)}{\leq} maxBBSize \\ &\quad \textbf{where}\ \ (m,\ \_) := protocolVersion \end{aligned}$$

$$\mathsf{lastAppliedHash} \in \mathsf{LastAppliedBlock}^? \to \mathsf{HashHeader}^?$$

$$\mathsf{lastAppliedHash}\ lab = \begin{cases} \Diamond & lab = \Diamond \\ h & lab = (\_,\ \_,\ h) \end{cases}$$

$$\mathsf{prtlSeqChecks} \in \mathsf{LastAppliedBlock}^? \to \mathsf{BHeader} \to \mathsf{Bool}$$

$$\mathsf{prtlSeqChecks}\ lab\ bh = \begin{cases} \mathsf{True} & lab = \Diamond \\ s_\ell \overset{(1)}{<} slot \wedge b_\ell + 1 \overset{(2)}{=} bn \wedge ph \overset{(3)}{=} \mathsf{hBPrevHeader}\ bhb & lab = (b_\ell,\ s_\ell,\ \_) \end{cases}$$

$$\begin{aligned} &\textbf{where} \\ &\quad bhb := \mathsf{headerBody}\ bh \\ &\quad bn := \mathsf{hBBlockNo}\ bhb \\ &\quad slot := \mathsf{hBSlot}\ bhb \\ &\quad ph := \mathsf{lastAppliedHash}\ lab \end{aligned}$$

Figure 17: Helper functions used in the Chain Head transition

$$
\text{ChainHead}\;
\frac{
\begin{array}{c}
bhb := \mathsf{headerBody}\ bh \qquad s := \mathsf{hBSlot}\ bhb \\[1ex]
\mathsf{prtlSeqChecks}\ lab\ bh \\[1ex]
\vdash nes \xrightarrow[\text{TICKF}]{s} forecast \\[1ex]
(e_1,\ \_,\ \_,\ \_,\ \_,\ \_) := nes \\
(e_2,\ \_,\ \_,\ es,\ \_,\ pd) := forecast \\
(\_,\ \_,\ \_,\ \_,\ pp) := es \\
ne := e_1 \neq e_2 \\
\eta_{ph} := \mathsf{prevHashToNonce}\ (\mathsf{lastAppliedHash}\ lab) \\[1ex]
\mathsf{chainChecks}\ \mathsf{MaxMajorPV}\ (\mathsf{maxHeaderSize}\ pp,\ \mathsf{maxBlockSize}\ pp,\ \mathsf{pv}\ pp)\ bh \\[1ex]
\begin{array}{l} \eta_c \\ \eta_{ph} \end{array} \vdash \begin{pmatrix} \eta_0 \\ \eta_h \end{pmatrix} \xrightarrow[\text{TICKN}]{ne} \begin{pmatrix} \eta_0' \\ \eta_h' \end{pmatrix} \\[1ex]
\begin{array}{l} pd \\ \eta_0' \end{array} \vdash \begin{pmatrix} cs \\ \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\text{PRTCL}]{bh} \begin{pmatrix} cs' \\ \eta_v' \\ \eta_c' \end{pmatrix} \\[1ex]
lab' := (\mathsf{hBBlockNo}\ bhb,\ s,\ \mathsf{headerHash}\ bh)
\end{array}
}{
nes \vdash \begin{pmatrix} cs \\ \eta_0 \\ \eta_v \\ \eta_c \\ \eta_h \\ lab \end{pmatrix} \xrightarrow[\text{CHAINHEAD}]{bh} \begin{pmatrix} cs' \\ \eta_0' \\ \eta_v' \\ \eta_c' \\ \eta_h' \\ lab' \end{pmatrix}
}
\quad (8)
$$

Figure 18: Chain Head rules

4 Properties

This section describes the properties that the consensus layer should have. The goal is to include these properties in the executable specification to enable e.g. property-based testing or formal verification.

4.1 Header-Only Validation

In any given chain state, the consensus layer needs to be able to validate the block headers without having to download the block bodies. Property 4.1 states that if an extension of a chain that spans less than StabilityWindow slots is valid, then validating the headers of that extension is also valid. This property is useful for its converse: if the header validation check for a sequence of headers does not pass, then we know that the block validation that corresponds to those headers will not pass either. In these properties, we refer to the CHAIN transition system as defined in [SL-D5].

Property 4.1 (Header only validation)**.** For all states s with slot number t[2], and chain extensions E with corresponding headers H such that:

$$0 \leq t_E - t \leq \mathsf{StabilityWindow}$$

we have:

$$\vdash s \xrightarrow[\text{CHAIN}]{E}{}^{*} s' \implies nes \vdash \tilde{s} \xrightarrow[\text{CHAINHEAD}]{H}{}^{*} \tilde{s}'$$

where $s = (nes,\ \tilde{s})$, t_E is the maximum slot number appearing in the blocks contained in E, and H is obtained from E by applying blockHeader to each block in E.

Property 4.2 (Body only validation)**.** For all states s with slot number t, and chain extensions $E = [b_0, \ldots, b_n]$ with corresponding headers $H = [h_0, \ldots, h_n]$ such that:

$$0 \leq t_E - t \leq \mathsf{StabilityWindow}$$

we have that for all $i \in [1, n]$:

$$nes \vdash \tilde{s} \xrightarrow[\text{CHAINHEAD}]{H}{}^{*} s_h \wedge \vdash (nes,\ \tilde{s}) \xrightarrow[\text{CHAIN}]{[b_0 \ldots b_{i-1}]}{}^{*} s_{i-1} \implies nes' \vdash \tilde{s}_{i-1} \xrightarrow[\text{CHAINHEAD}]{h_i} s'_h$$

where $s = (nes,\ \tilde{s})$, $s_{i-1} = (nes',\ \tilde{s}_{i-1})$, t_E is the maximum slot number appearing in the blocks contained in E.

Property 4.2 states that if we validate a sequence of headers, we can validate their bodies independently and be sure that the blocks will pass the chain validation rule. To see this, given an environment e and initial state s, assume that a sequence of headers $H = [h_0, \ldots, h_n]$ corresponding to blocks in $E = [b_0, \ldots, b_n]$ is valid according to the CHAINHEAD transition system:

$$nes \vdash \tilde{s} \xrightarrow[\text{CHAINHEAD}]{H}{}^{*} \tilde{s}'$$

Assume the bodies of E are valid according to the BBODY rules (defined in [SL-D5]), but E is not valid according to the CHAIN rule. Assume that there is a $b_j \in E$ such that it is **the first block** such that does not pass the CHAIN validation. Then:

$$\vdash (nes,\ \tilde{s}) \xrightarrow[\text{CHAIN}]{[b_0, \ldots b_{j-1}]}{}^{*} s_j$$

[2] i.e. the component s_ℓ of the last applied block of s equals t

But by Property 4.2 we know that

$$nes_j \vdash \tilde{s}_j \xrightarrow[\text{CHAINHEAD}]{h_j} \tilde{s}_{j+1}$$

which means that block b_j has valid headers, and this in turn means that the validation of b_j according to the chain rules must have failed because it contained an invalid block body. But this contradicts our assumption that the block bodies were valid.

Values associated with the leader value calculations

$certNat \in \{n \mid n \in \mathbb{N}, n \in [0, 2^{512})\}$	Certified natural value from VRF
$f \in [0, 1]$	Active slot coefficient
$\sigma \in [0, 1]$	Stake proportion

5 Leader Value Calculation

This section details how we determine whether a node is entitled to lead (under the Praos protocol) given the output of its verifiable random function calculation.

5.1 Computing the leader value

The verifiable random function gives us a 64-byte random output. We interpret this as a natural number *certNat* in the range $[0, 2^{512})$.

5.2 Node eligibility

As per [DGKR17], a node is eligible to lead when its leader value $p < 1 - (1 - f)^{\sigma}$. We have

$$\begin{aligned} p &< 1 - (1 - f)^{\sigma} \\ \iff \left(\frac{1}{1-p}\right) &< \exp\left(-\sigma \cdot \ln\left(1 - f\right)\right) \end{aligned}$$

The latter inequality can be efficiently computed through use of its Taylor expansion and error estimation to stop computing terms once we are certain that the result will be either above or below the target value.

We carry out all computations using fixed precision arithmetic (specifically, we use 34 decimal bits of precision, since this is enough to represent the fraction of a single lovelace.)

As such, we define the following:

$$\begin{aligned} p &= \frac{certNat}{2^{512}} \\ q &= 1 - p \\ c &= \ln\left(1 - f\right) \end{aligned}$$

and define the function *checkLeaderVal* as follows:

$$\mathsf{checkLeaderVal}\ certNat\ \sigma\ f = \begin{cases} \mathsf{True}, & f = 1 \\ \frac{1}{q} < \exp\left(-\sigma \cdot c\right), & \text{otherwise} \end{cases}$$

References

[BC-D1] IOHK Formal Methods Team. Byron Blockchain Specification, IOHK Deliverable BC-D1, 2019. URL https://github.com/intersectmbo/cardano-ledger/tree/master/docs/.

[SL-D5] IOHK Formal Methods Team. A Formal Specification of the Cardano Ledger, 2019. URL https://github.com/intersectmbo/cardano-ledger/releases/latest/download/shelley-ledger.pdf

[DGKR17] B. M. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. *IACR Cryptology ePrint Archive*, 2017:573, 2017.

[SL-D1] IOHK Formal Methods Team. Design Specification for Delegation and Incentives in Cardano, IOHK Deliverable SL-D1, 2018. URL https://github.com/intersectmbo/cardano-ledger/releases/latest/download/shelley-delegation.pdf.

A Cryptographic Details

A.1 Abstract functions

- The nonce operation $x \star y$ from Figure 3 is implemented as the BLAKE2b-256 hash of the concatenation of x and y.
- The functions slotToSeed and nonceToSeed from Figure 3 are implemented as the big-endian encoding of the slot/nonce number in 8 bytes.