Specification of the Blockchain Layer

Marko Dimjašević
Nicholas Clarke

May 2, 2019

Abstract

This documents defines inference rules for operations on a blockchain as a specification of the blockchain layer of Cardano in the Byron release and in a transition to the Shelley release. In particular, a block validity definition is given, which is accompanied by small-step operational semantics inference rules.

List of Contributors

Damian Nadales, Yun Lu.

Contents

List of Figures

1 Introduction

The idea behind this document is to formalise what it means for a new block to be added to the blockchain to be valid. The scope of the document is the Byron release and a transition phase to the Shelley release of the Cardano blockchain platform.

Unless a new block is valid, it cannot be added to the blockchain and thereby extend it. This is needed for a system that is subscribed to the blockchain and keeps a copy of it locally. In particular, this document gives a formalisation that should be straightforward to implement in a programming language, e.g., in Haskell.

This document is intended to be read in conjunction with Damian Nadales (2019), which covers the payload carried around in the blockchain. Certain of the underlying systems and types defined will rely on definitions in that document.

2 Preliminaries

Powerset Given a set X, $\mathbb{P}\,\mathsf{X}$ is the set of all the subsets of X.

Sequence Given a set X, X^* is a sequence having elements taken from X. The empty sequence is denoted by ϵ, and given a sequence Λ, $\Lambda; x$ is the sequence that results from appending $x \in \mathsf{X}$ to Λ. Furthermore, ϵ is an identity element for sequence joining: $\epsilon; x = x; \epsilon = x$.

Dropping on sequences Given a sequence Λ, $\Lambda \downarrow n$ is the sequence that is obtained after removing (dropping) the first n elements from Λ. If $n \leq 0$ then $\Lambda \downarrow n = \Lambda$.

$$
\begin{aligned}
K \lhd M &= \{i \mapsto o \mid i \mapsto o \in M,\ i \in K\} && \text{domain restriction} \\
K \lhd\!\!\!\!/\ M &= \{i \mapsto o \mid i \mapsto o \in M,\ i \notin K\} && \text{domain exclusion} \\
M \rhd V &= \{i \mapsto o \mid i \mapsto o \in M,\ o \in V\} && \text{range restriction} \\
\underrightarrow{\bigcup} &\in (A \mapsto B) \to (A \mapsto B) \to (A \mapsto B) && \text{union override} \\
d_0 \underrightarrow{\bigcup} d_1 &= d_1 \cup (\operatorname{dom} d_1 \lhd\!\!\!\!/\ d_0)
\end{aligned}
$$

Figure 1: Definition of the Union Override Operation

Appending with a moving window Given a sequence Λ, we define

$$\Lambda ;_w x := (\Lambda ; x) \downarrow (|\Lambda| + 1 - w)$$

Filtering on sequences Given a sequence Λ, and a predicate p on the elements of Λ, filter $p\ \Lambda$ is the sequence that contains all the elements of Λ that satisfy p, in the same order they appear on Λ.

Option type An option type in type A is denoted as $A^? = A + \diamond$. The A case corresponds to a case when there is a value of type A and the $\diamond$ case corresponds to a case when there is no value.

Union override The union override operation is defined in Figure 1.

Pattern matching in premises In the inference-rules premises use $patt = exp$ to pattern-match an expression exp with a certain pattern $patt$. For instance, we use $\Lambda' ; x = \Lambda$ to be able to deconstruct a sequence Λ in its last element, and prefix. If an expression does not match the given pattern, then the premise does not hold, and the rule cannot trigger.

Maps and partial functions $A \mapsto B$ denotes a **partial function** from A to B, which can be seen as a map (dictionary) with keys in A and values in B. Given a map $m \in A \mapsto B$, notation $a \mapsto b \in m$ is equivalent to $m\ a = b$.

2.1 Sets

There are several standard sets used in the document:

Booleans The set of booleans is denoted with $\mathbb{B}$ and has two values, $\mathbb{B} = \{\bot, \top\}$.

Natural numbers The set of natural numbers is denoted with $\mathbb{N}$ and defined as $\mathbb{N} = \{0, 1, 2, \dots\}$.

3 Update interface

We define a general update interface to abstract over the various update state transitions which happen when a new block is processed. Figure 2 defines the type of signals used for this system. Figure 4 defines the rules for this system. The two rules handle the cases where there is or is not an update proposal contained within the block.

Update interface signals

$$\mathsf{UpdatePayload} = \left(\begin{array}{rcll} \mathit{mprop} & \in & \mathsf{UProp}^? & \text{possible update proposal} \\ \mathit{votes} & \in & \mathsf{Vote}^* & \text{votes for update proposals} \\ \mathit{end} & \in & (\mathsf{VKey} \times \mathsf{ProtVer}) & \text{protocol version endorsment} \end{array} \right)$$

Figure 2: Update interface processing types and functions

Update interface processing transitions

$$\_ \vdash \_ \xrightarrow[\text{BUPI}]{} \_ \subseteq \mathbb{P}\,(\mathsf{UPIEnv} \times \mathsf{UPIState} \times \mathsf{UpdatePayload} \times \mathsf{UPIState})$$

Figure 3: Update interface processing transition-system types

$$\frac{\Gamma \vdash \mathit{us} \xrightarrow[\text{UPIREG}]{\mathit{prop}} \mathit{us}' \qquad \Gamma \vdash \mathit{us}' \xrightarrow[\text{UPIVOTES}]{\mathit{votes}} \mathit{us}'' \qquad \Gamma \vdash \mathit{us}'' \xrightarrow[\text{UPIEND}]{\mathit{end}} \mathit{us}'''}{\Gamma \vdash \mathit{us} \xrightarrow[\text{BUPI}]{\left(\begin{array}{l} \mathit{prop} \\ \mathit{votes} \\ \mathit{end} \end{array} \right)} \mathit{us}'''}$$

$$\frac{\Gamma \vdash \mathit{us} \xrightarrow[\text{UPIVOTES}]{\mathit{votes}} \mathit{us}' \qquad \Gamma \vdash \mathit{us}' \xrightarrow[\text{UPIEND}]{\mathit{end}} \mathit{us}''}{\Gamma \vdash \mathit{us} \xrightarrow[\text{BUPI}]{\left(\begin{array}{l} \diamond \\ \mathit{votes} \\ \mathit{end} \end{array} \right)} \mathit{us}''}$$

Figure 4: Update interface processing rules

4 Permissive BFT

The majority of this specification is concerned with the processing of the *ledger*; that is, the content (contained in both the block header and the block body). In addition, however, we must also concern ourselves with the protocol used to transmit the blocks and whether, according to that protocol, we may validly extend the chain with a new block (assuming that block forms a valid extension to the chain under the ledger rules).

Cardano's planned evolution can be split into roughly three eras:

Byron/Ouroboros In the Byron/Ouroboros era, the Ouroboros (Kiayias et al. (2017)) protocol is used to control who is eligible to issue a block, using a stake distribution mediated by heavyweight delegation certificates. The Byron payload includes such things as VSS and payloads verified by individual signatures.

Handover In the handover era, blocks will be issued according to Ouroboros BFT (Kiayias and Russell (2018)). The Byron payload will be retained, although parts of will be superfluous.

Shelley/Praos In the Shelley/Praos era, blocks will be issued according to the Ouroboros Praos (David et al. (2017)) protocol, with stake distribution determined according to the new delegation design in Kant et al. (2018).

During the handover era (as described in this document), while blocks will be issued according to Ouroboros BFT, they will be validated according to a variant known as Permissive BFT. This is designed such that it will successfully validate blocks issued both under Ouroboros and under Ouroboros BFT (with a high probability - see Appendix A).

This section therefore will describe the section of the rules concerned with the Permissive BFT protocol. Note that all of these are concerned only with the block header, since the block body is entirely concerned with the ledger.

4.1 Counting signed blocks

To guard against the compromise of a minority of the genesis keys, we require that in the rolling window of the last k blocks, where k is the chain stability parameter, the number of blocks signed by keys that sk_s delegated to is no more than a threshold $k \cdot t$, where t is a constant that will be picked in the range $1/5 \leq t \leq 1/4$. Initial research suggests setting $t = 0.22$ as a good value. Specifically, given $k = 2160$, we would allow a single genesis key to issue (via delegates) 475 blocks (since $2160 \cdot 0.22 = 475.2$), but a 476^{th} block would be rejected. See Appendix A for the background on this value. The abstract constant (nullary functions) related to the protocol are defined in Figure 5.

Abstract functions

k	$\in \mathbb{N}$	chain stability parameter
t	$\in \left[\frac{1}{5}, \frac{1}{4}\right]$	block signature count threshold
dms	$\in \mathsf{DIState} \to (\mathsf{VKey_G} \mapsto \mathsf{VKey})$	delegation-state delegation-map

Figure 5: Protocol abstract functions

Figure 7 gives the rules for signature counting. We verify that the key that delegates to the signer of this block has not already signed more than its allowed threshold of blocks. If there are no delegators for the given key, or if there is more than one delegator, the rule will fail to trigger. We then update the sequence of signers, and drop those elements that fall outside the size of the moving window (k).

Block signature count environments

$$\mathsf{BSCEnv} = \left(\; ds \; \in \; \mathsf{DIState} \quad \text{delegation state} \; \right)$$

Block signature count transitions

$$\_ \vdash \_ \xrightarrow[\text{SIGCNT}]{\_} \_ \subseteq \mathbb{P} \left(\mathsf{BSCEnv} \times {\mathsf{VKey_G}}^* \times \mathsf{VKey} \times {\mathsf{VKey_G}}^* \right)$$

Figure 6: Block signature count transition-system types

$$\frac{\{vk_g\} := \mathrm{dom}\left((\mathsf{dms}\; ds) \rhd \{vk_d\}\right) \qquad sgs' := sgs ;_k vk_g \qquad \left|\mathsf{filter}\; (= vk_g)\; sgs'\right| \leq k \cdot t}{ds \vdash sgs \xrightarrow[\text{SIGCNT}]{vk_d} sgs'}$$

Figure 7: Block signature count rules

4.2 Permissive BFT Header Processing

During PBFT processing of the block header, we do the following:

1. We check that the current block is being issued for a slot later than that of the last block issued.
2. We check that the current block is being issued for a slot no later than the current slot (as determined by the system clock).
3. We check that the previous block hash contained in the block header corresponds to the known hash of the previous block.
4. We check that the header signature correctly verifies the "signed" content of the header.
5. Finally, we verify and update the state according to the signature count rules.

Abstract types

$$bh \in \mathsf{BlockHeader} \quad \text{Block header}$$

Abstract functions

$$\begin{array}{rcll} \mathsf{bhPrevHash} & \in & \mathsf{BlockHeader} \to \mathsf{Hash} & \text{previous header hash} \\ \mathsf{bhHash} & \in & \mathsf{BlockHeader} \to \mathsf{Hash} & \text{header hash} \\ \mathsf{bhSig} & \in & \mathsf{BlockHeader} \to \mathsf{Sig} & \text{block signature} \\ \mathsf{bhIssuer} & \in & \mathsf{BlockHeader} \to \mathsf{VKey} & \text{block issuer} \\ \mathsf{bhSlot} & \in & \mathsf{BlockHeader} \to \mathsf{Slot} & \text{slot for which this block is issued} \end{array}$$

Figure 8: Permissive BFT types and functions

Permissive BFT environments

$$\mathsf{PBFTEnv} = \left(\begin{array}{rcll} ds & \in & \mathsf{DIState} & \text{delegation state} \\ s_{last} & \in & \mathsf{Slot} & \text{slot for which the last known block was issued} \\ s_{now} & \in & \mathsf{Slot} & \text{current slot} \end{array} \right)$$

Permissive BFT states

$$\mathsf{PBFTState} = \left(\begin{array}{rcll} h & \in & \mathsf{Hash} & \text{Tip header hash} \\ sgs & \in & \mathsf{VKey_G}^* & \text{Last signers} \end{array} \right)$$

Permissive BFT transitions

$$\_ \vdash \_ \xrightarrow[\text{PBFT}]{} \_ \subseteq \mathbb{P}\left(\mathsf{PBFTEnv} \times \mathsf{PBFTState} \times \mathsf{BlockHeader} \times \mathsf{PBFTState}\right)$$

Figure 9: Permissive BFT transition-system types

$$\frac{\begin{array}{cc} vk_d := \mathsf{bhIssuer}\ bh & s := \mathsf{bhSlot}\ bh \\ s > s_{last} & s \leq s_{now} \\ \mathsf{bhPrevHash}\ bh = h & \mathsf{verify}\ vk_d\ [\![\mathsf{bhToSign}\ bh]\!]\ (\mathsf{bhSig}\ bh) \end{array} \qquad ds \vdash sgs \xrightarrow[\text{SIGCNT}]{vk_d} sgs'}{\begin{pmatrix} ds \\ s_{last} \\ s_{now} \end{pmatrix} \vdash \begin{pmatrix} h \\ sgs \end{pmatrix} \xrightarrow[\text{PBFT}]{bh} \begin{pmatrix} \mathsf{bhHash}\ bh \\ sgs' \end{pmatrix}}$$

Figure 10: Permissive BFT rules

5 Epoch transitions

During each block transition, we must determine whether that block sits on an epoch boundary and, if so, carry out various actions which are done on that boundary. In the BFT era, the only computation carried out at the epoch boundary is the update of protocol versions.

We rely on a function sEpoch, whose type is given in Figure 11, to determine the epoch corresponding to a given slot. We do not provide an implementation for such function in this specification, but in practice a possible way of implementing such function is to rely on map from the epochs to their corresponding length (given in number of slots they contain). Such a map would also be required by the database layer to find the requisite epoch file to look up a given block. We envision that an implementation may of course choose a more compact representation for this partial function that only records the changes in epoch length, rather than storing a length for each epoch. In addition, we rely on abstract constant (nullary function) *ngk*, which determines the number of genesis keys.

It is also worth noticing that in the Byron era, the number of slots per-epoch is fixed to $10 \cdot k$, where k is the chain stability parameter.

Figure 13 determines when an epoch change has occurred and updates the update state to the correct version.

Abstract functions

$$\begin{array}{rcll} \mathsf{sEpoch} & \in & \mathsf{Slot} \to \mathbb{N} \to \mathsf{Epoch} & \text{epoch containing this slot} \\ \mathit{ngk} & \in & \mathbb{N} & \text{number of genesis keys} \end{array}$$

Figure 11: Epoch transition types and functions

Epoch transition environments

$$\mathsf{ETEnv} = \left(\; e_c \; \in \; \mathsf{Epoch} \quad \text{current epoch} \; \right)$$

Epoch transition states

$$\mathsf{ETState} = \left(\; us \; \in \; \mathsf{UPIState} \quad \text{update interface state} \; \right)$$

Epoch transition transitions

$$\_ \vdash \_ \xrightarrow[\text{EPOCH}]{\_} \_ \subseteq \mathbb{P} \left(\mathsf{ETEnv} \times \mathsf{ETState} \times \mathsf{Slot} \times \mathsf{ETState} \right)$$

Figure 12: Epoch transition transition-system types

$$\frac{e_c \geq \mathsf{sEpoch}\ s\ k}{e_c \vdash \; us \xrightarrow[\text{EPOCH}]{s} us}$$

$$\frac{e_c < \mathsf{sEpoch}\ s\ k \qquad \mathsf{sEpoch}\ s\ k \vdash us \xrightarrow[\text{UPIEC}]{} us'}{e_c \vdash \; us \xrightarrow[\text{EPOCH}]{s} us'}$$

Figure 13: Epoch transition rules

6 Block processing

We delineate here between processing the header and body of a block. It's useful to make this distinction since we may process headers ahead of the block body, and we have less context available to process headers - in particular, we must be able to process block headers without the recent history of block bodies.

Abstract types

$$\begin{array}{rcll} b & \in & \mathsf{Block} & \text{block} \\ h & \in & \mathsf{Hash} & \text{hash} \\ data & \in & \mathsf{Data} & \text{data} \end{array}$$

Abstract functions

$$\begin{array}{rcll} \mathsf{bSize} & \in & \mathsf{Block} \to \mathbb{N} & \text{block size in bytes} \\ \mathsf{verify} & \in & \mathsf{VKey} \times \mathsf{Data} \times \mathsf{Sig} & \text{verification relation} \end{array}$$

Figure 14: Basic Block-related Types and Functions

6.1 Block header processing

Processing headers doesn't require any changes to the state, so we simply check predicates. Figure 1 gives the validity predicate for a header. We verify that the block header does not exceed the maximum size specified in the protocol parameters. The maxHeaderSize protocol parameter is defined in Damian Nadales (2019).

Abstract types

$$\begin{array}{rcll} bh & \in & \mathsf{BlockHeader} & \text{block header} \\ bts & \in & \mathsf{BHToSign} & \text{part of the block header which must be signed} \end{array}$$

Abstract functions

$$\begin{array}{rcll} \mathsf{bHead} & \in & \mathsf{Block} \to \mathsf{BlockHeader} & \text{block header} \\ \mathsf{bHeaderSize} & \in & \mathsf{BlockHeader} \to \mathbb{N} & \text{block header size in bytes} \end{array}$$

Figure 15: Block header processing types and functions

$$\mathsf{headerIsValid}\ us\ bh = \mathsf{maxHeaderSize} \mapsto s_{max} \in \mathsf{pps}\ us \Rightarrow \mathsf{bHeaderSize}\ bh \leq s_{max} \tag{1}$$

Figure 16: Block header validity functions

6.2 Block body processing

During processing of the block body, we perform two main functions: verification of the body integrity using the proofs contained in the block header, and update of the various state components. These rules are given in Figure 19, where the types and the functions used there are defined in Figure 18. The UTxO, delegation, and update state as well as the maxBlockSize protocol parameter are defined in Damian Nadales (2019).

Verification is done independently for the three components of the body payload: UTxO, delegation and update. Each of these three has a hash in the block header. Note that Byron-era block payload also has an additional component: the VSS payload. This part of the block is unnecessary during the BFT era, and hence we do not verify it.

In addition to block verification, we also process the three components of the payload; UTxO, delegation and update.

Abstract functions

bUtxo	$\in \mathsf{Block} \to (\mathsf{Tx} \times \mathbb{P}\ (\mathsf{VKey} \times \mathsf{Sig}))$	block UTxO payload
bCerts	$\in \mathsf{Block} \to \mathsf{DCert}^*$	block certificates
$\mathsf{bUpdProp}$	$\in \mathsf{Block} \to \mathsf{UProp}^?$	block update proposal payload
$\mathsf{bUpdVotes}$	$\in \mathsf{Block} \to \mathsf{Vote}^*$	block update votes payload
$\mathsf{bProtVer}$	$\in \mathsf{Block} \to \mathsf{ProtVer}$	block protocol version
$\mathsf{bhUtxoHash}$	$\in \mathsf{BlockHeader} \to \mathsf{Hash}$	UTxO payload hash
$\mathsf{bhDlgHash}$	$\in \mathsf{BlockHeader} \to \mathsf{Hash}$	delegation payload hash
$\mathsf{bhUpdHash}$	$\in \mathsf{BlockHeader} \to \mathsf{Hash}$	update payload hash
hash	$\in \mathsf{Data} \to \mathsf{Hash}$	hash function

Derived functions

$\mathsf{bEndorsment}$	$\in \mathsf{Block} \to \mathsf{ProtVer} \times \mathsf{VKey}$	Protocol version endorsment
$\mathsf{bEndorsment}\ b$	$= (\mathsf{bProtVer}\ b, (bhIssuer \cdot bHead)\ b)$	
bSlot	$\in \mathsf{Block} \to \mathsf{Slot}$	Slot for which this block is being issued
$\mathsf{bSlot}\ b$	$= (bhSlot \cdot bHead)\ b$	
$\mathsf{bUpdPayload}$	$\in \mathsf{Block} \to (\mathsf{UProp}^? \times \mathsf{Vote}^*)$	Block update payload
$\mathsf{bUpdPayload}\ b$	$= (\mathsf{bUpdProp}\ b,\ \mathsf{bUpdVotes}\ b)$	

Figure 17: Block body processing types and functions

Block body processing environments

$$\mathsf{BBEnv} = \left(\begin{array}{rcll} pps & \in & \mathsf{PParams} & \text{protocol parameters} \\ e_n & \in & \mathsf{Epoch} & \text{epoch we are currently processing blocks for} \\ utxo_0 & \in & \mathsf{UTxO} & \text{genesis UTxO} \end{array} \right)$$

Block body processing states

$$\mathsf{BBState} = \left(\begin{array}{rcll} utxoSt & \in & \mathsf{UTxOState} & \text{UTxO state} \\ ds & \in & \mathsf{DIState} & \text{delegation state} \\ us & \in & \mathsf{UPIState} & \text{update interface state} \end{array} \right)$$

Block body processing transitions

$$\_ \vdash \_ \xrightarrow[\text{BBODY}]{} \_ \subseteq \mathbb{P}\ (\mathsf{BBEnv} \times \mathsf{BBState} \times \mathsf{Block} \times \mathsf{BBState})$$

Figure 18: Block body processing transition-system types

$$\frac{\begin{array}{c} \mathsf{maxBlockSize} \mapsto b_{max} \in pps \qquad \mathsf{bSize}\ b \leq b_{max} \\ bh := \mathsf{bHead}\ b \qquad vk_d := \mathsf{bIssuer}\ bh \\ \mathsf{hash}\ (\mathsf{bUtxo}\ b) = \mathsf{bhUtxoHash}\ bh \qquad \mathsf{hash}\ (\mathsf{bCerts}\ b) = \mathsf{bhDlgHash}\ bh \\ \mathsf{hash}\ (\mathsf{bUpdPayload}\ b) = \mathsf{bhUpdHash}\ bh \\ \left(\begin{array}{l} \mathsf{bSlot}\ b \\ \mathsf{dms}\ ds \end{array} \right) \vdash us \xrightarrow[\text{BUPI}]{\left(\begin{array}{l} \mathsf{bUpdProp}\ b \\ \mathsf{bUpdVotes}\ b \\ \mathsf{bEndorsment}\ b \end{array} \right)} us' \\ \left(\begin{array}{l} \mathsf{dom}\ (\mathsf{dms}\ ds) \\ e_n \\ \mathsf{bSlot}\ b \end{array} \right) \vdash ds \xrightarrow[\text{DELEG}]{\mathsf{bCerts}\ b} ds' \qquad \left(\begin{array}{l} utxo_0 \\ pps \end{array} \right) \vdash utxoSt \xrightarrow[\text{UTXOWS}]{\mathsf{bUtxo}\ b} utxoSt' \end{array}}{\left(\begin{array}{l} pps \\ e_n \\ utxo_0 \end{array} \right) \vdash \left(\begin{array}{c} utxoSt \\ ds \\ us \end{array} \right) \xrightarrow[\text{BBODY}]{b} \left(\begin{array}{c} utxoSt' \\ ds' \\ us' \end{array} \right)}$$

Figure 19: Block body processing rules

7 Blockchain extension

Figure 22 captures the central chain extension rule. This has two variants, depending on whether the block in question is an epoch boundary block. Epoch boundary blocks are not required during the BFT era, but whilst they are not distributed, epoch boundary blocks must still be processed since their hash forms part of the chain. Since we do not care about the contents of an epoch boundary block, we check that it does not exceed some suitably large size, and otherwise simply update the header hash to the block hash.

If the block is not an epoch boundary block, then we process:

- a potential epoch change according to the rules in figure 13,
- the header using the validity predicate of equation 1, and
- the body according to the rules in figure 19.

Abstract functions

$$\begin{array}{rcll} \mathsf{bIsEBB} & \in & \mathsf{Block} \to \mathbb{B} & \text{epoch boundary block check} \\ \mathsf{pps} & \in & \mathsf{UPIState} \to \mathsf{PParams} & \text{update-state protocol-parameters} \end{array}$$

Figure 20: Blockchain Extension Types and Functions

Chain extension environments

$$\mathsf{CEEnv} = \left(\begin{array}{rcll} s_{now} & \in & \mathsf{Slot} & \text{current slot} \\ utxo_0 & \in & \mathsf{UTxO} & \text{genesis UTxO} \end{array} \right)$$

Chain extension states

$$\mathsf{CEState} = \left(\begin{array}{rcll} s_{last} & \in & \mathsf{Slot} & \text{slot of the last seen block} \\ sgs & \in & \mathsf{VKey_G}^* & \text{last signers} \\ h & \in & \mathsf{Hash} & \text{current block hash} \\ utxoSt & \in & \mathsf{UTxO} & \text{UTxOState} \\ ds & \in & \mathsf{DIState} & \text{delegation state} \\ us & \in & \mathsf{UPIState} & \text{update interface state} \end{array} \right)$$

Chain extension transitions

$$\_ \vdash \_ \xrightarrow[\text{CHAIN}]{} \_ \subseteq \mathbb{P}\left(\mathsf{CEEnv} \times \mathsf{CEState} \times \mathsf{Block} \times \mathsf{CEState}\right)$$

Figure 21: Blockchain extension transition-system types

$$\frac{\mathsf{bIsEBB}\ b \qquad \mathsf{bSize}\ b \leq 2^{21} \qquad h' := \mathsf{bhHash}\ (\mathsf{bHead}\ b)}{\begin{pmatrix} s_{now} \\ utxo_0 \end{pmatrix} \vdash \begin{pmatrix} s_{last} \\ sgs \\ h \\ utxoSt \\ ds \\ us \end{pmatrix} \xrightarrow[\text{CHAIN}]{b} \begin{pmatrix} s_{last} \\ sgs \\ h' \\ utxoSt \\ ds \\ us \end{pmatrix}}$$

$$\frac{\neg\mathsf{bIsEBB}\ b}{\left(\ \mathsf{sEpoch}\ s_{last}\ k\ \right) \vdash \left(\ us\ \right) \xrightarrow[\text{EPOCH}]{\mathsf{bSlot}\ b} \left(\ us'\ \right)}$$

$$\frac{\mathsf{headerIsValid}\ us'\ (\mathsf{bHead}\ b)}{\begin{pmatrix} \mathsf{pps}\ us' \\ \mathsf{sEpoch}\ (\mathsf{bSlot}\ b)\ k \\ utxo_0 \end{pmatrix} \vdash \begin{pmatrix} utxoSt \\ ds \\ us' \end{pmatrix} \xrightarrow[\text{BBODY}]{b} \begin{pmatrix} utxoSt' \\ ds' \\ us'' \end{pmatrix}}$$

$$\frac{\begin{pmatrix} ds \\ s_{last} \\ s_{now} \end{pmatrix} \vdash \begin{pmatrix} h \\ sgs \end{pmatrix} \xrightarrow[\text{PBFT}]{\mathsf{bHead}\ b} \begin{pmatrix} h' \\ sgs' \end{pmatrix}}{\begin{pmatrix} s_{now} \\ utxo_0 \end{pmatrix} \vdash \begin{pmatrix} s_{last} \\ sgs \\ h \\ utxoSt \\ ds \\ us \end{pmatrix} \xrightarrow[\text{CHAIN}]{b} \begin{pmatrix} \mathsf{bSlot}\ b \\ sgs' \\ h' \\ utxoSt' \\ ds' \\ us'' \end{pmatrix}}$$

Figure 22: Blockchain extension rules

8 Transition systems properties

8.1 Header only validation

The following transition system is used in the properties enunciated in this section.

Definition 1 (EPOCH+BHEAD+PBFT STS)

$$\left(\begin{array}{l} \mathsf{sEpoch}\ s_{last}\ k \end{array} \right) \vdash \left(\begin{array}{l} us \end{array} \right) \xrightarrow[\text{EPOCH}]{\mathsf{bSlot}\ b} \left(\begin{array}{l} us' \end{array} \right)$$

$$\mathsf{headerIsValid}\ us'\ (\mathsf{bHead}\ b)$$

$$\frac{\begin{pmatrix} ds \\ s_{last} \\ s_{now} \end{pmatrix} \vdash \begin{pmatrix} h \\ sgs \end{pmatrix} \xrightarrow[\text{PBFT}]{bh} \begin{pmatrix} h' \\ sgs' \end{pmatrix}}{\begin{pmatrix} ds \\ s_{last} \\ s_{now} \end{pmatrix} \vdash \begin{pmatrix} h \\ sgs \\ us \end{pmatrix} \xrightarrow[\text{EPOCH+BHEAD+PBFT}]{bh} \begin{pmatrix} h' \\ sgs' \\ us' \end{pmatrix}}$$

In any given ledger state, the consensus layer needs to be able to validate the block headers without having to download the block bodies. Property 1 states that if an extension of a chain that spans less than $2 \cdot k$ 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.

Property 1 (Header only validation) *For all environments e, states s with slot number t[1], and chain extensions E with corresponding headers H such that:*

$$0 \leq t_E - t \leq 2 \cdot k$$

we have:

$$e \vdash s \xrightarrow[\text{CHAIN}]{E}{}^{*} s' \implies e_h \vdash s_h \xrightarrow[\text{EPOCH+BHEAD+PBFT}]{H}{}^{*} s'_h$$

where t_E is the maximum slot number appearing in the blocks contained in E, $e_h := \mathsf{h_e}\ e\ s$ and $s_h := \mathsf{h_s}\ e\ s$, and functions $\mathsf{h_e}$ and $\mathsf{h_s}$ select the appropriate environment and state components needed by the EPOCH+BHEAD+PBFT *transition system in the obvious way.*

Property 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 EPOCH+BHEAD+PBFT transition system:

$$e_h \vdash s_h \xrightarrow[\text{EPOCH+BHEAD+PBFT}]{H}{}^{*} s'_h$$

where e_h and s_h are obtained from e and s as described in Property 1. Assume the bodies of E are valid according to the BBODY rules, 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:

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

[1] i.e. the component s_{last} of s equals t

But by Property 2 we know that

$$e_{h_j} \vdash s_{h_j} \xrightarrow[\text{EPOCH+BHEAD+PBFT}]{h_j} s_{h_{j+1}}$$

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.

Property 2 (Body only validation) *For all environments e, states s with slot number t, and chain extensions $E = [b_0, \ldots, b_n]$ with corresponding headers H such that:*

$$0 \leq t_E - t \leq 2 \cdot k$$

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

$$e_h \vdash s_h \xrightarrow[\text{EPOCH+BHEAD+PBFT}]{H}{}^* s'_h \wedge e \vdash s \xrightarrow[\text{CHAIN}]{[b_0 \ldots b_{i-1}]}{}^* s_{i-1} \implies e_{h_{i-1}} \vdash s_{h_{i-1}} \xrightarrow[\text{EPOCH+BHEAD+PBFT}]{h_i} s''_{h_i}$$

where t_E is the maximum slot number appearing in the blocks contained in E, $e_h := \mathsf{h_e}\ e\ s$ and $s_h := \mathsf{h_s}\ e\ s$, $e_{h_{i-1}} := \mathsf{h_e}\ e\ s_{i-1}$ and $s_{h_{i-i}} := \mathsf{h_s}\ e\ s_{i-1}$, and $\mathsf{h_e}$ and $\mathsf{h_s}$ are the same functions mentioned in Property 1.

Property 3 expresses the fact the there is a function that allow us to recover the header-only state by rolling back at most k blocks, and use this state to validate the headers of an alternate chain. Note that this property is not inherent to the CHAIN rules and can be trivially satisfied by any function that keeps track of the history of the intermediate chain states up to k blocks back. This property is stated here so that it can be used as a reference for the tests in the consensus layer, which uses the rules presented in this document.

Property 3 (Existence of roll back function) *There exists a function f such that for all chains*

$$C = C_0; b; C_1$$

we have that if for all alternative chains C'_1, $|C'_1| \leq k$, with corresponding headers H'_1

$$e \vdash s_0 \xrightarrow[\text{CHAIN}]{C_0;b}{}^* s_1 \xrightarrow[\text{CHAIN}]{C_1}{}^* s_2 \wedge e \vdash s_1 \xrightarrow[\text{CHAIN}]{C'_1}{}^* s'_1 \implies (\mathsf{f}\ (\mathsf{bHead}\ b)\ s_2) \xrightarrow[\text{EPOCH+BHEAD+PBFT}]{H'_1}{}^* s_h$$

References

IOHK Damian Nadales. A formal specification of the cardano ledger, 2019. URL https://hydra.iohk.io/job/Cardano/cardano-ledger/byronLedgerSpec/latest/download/1/ledger-spec.pdf.

Bernardo David, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol, 2017. URL https://eprint.iacr.org/2017/573.pdf.

Philipp Kant, Lars Brünjes, and Duncan Coutts. Design specification for delegation and incentives in cardano, 2018. URL https://github.com/input-output-hk/cardano-ledger/tree/master/docs/delegation_design_spec.

Aggelos Kiayias and Alexander Russell. Ouroboros-bft: A simple byzantine fault tolerant consensus protocol, 2018. URL https://api.zotero.org/groups/478201/items/J6E2Q8L7/file/view?key=Qcjdk4erSuUZ8jvAah59Asef.

Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol, 2017. URL https://eprint.iacr.org/2016/889.pdf.

Appendices

A Calculating the t parameter

We originally give the range of t as between $\frac{1}{5}$ and $\frac{1}{4}$. The upper bound here is to reduce the possible number of malicious blocks; if two of the genesis keys are comprimised, the attackers may not be able to produce a longer chain than the honest participants. The lower bound is required to prevent a situation in which a chain produced under the initial Ouroboros setting produces a chain which, according to the new BFT semantics, is invalid.

In order to determine the best value of t, we must consider the likelihood of such an invalid chain being produced by the old procedure of randomly selecting the slot leaders for each slot. Given the Cardano chain is still federated, the likelihood of this happening is the same for each of the 7 stakeholders, and we may model the number of selected slots within a k-slot window X as a binomial distribution $X \sim \mathrm{B}\left(k, \frac{1}{7}\right)$.

In each epoch of size n blocks, there are $n - k + 1$ such k-block windows. Boole's inequality gives us that the likelihood of exceeding the threshold in any one of these windows is bounded above by the sum of the likelihoods for each window. We may thus consider that the probability of a given stakeholder violating the threshold in an epoch to be bounded by

$$(n - k + 1) \cdot P(X > t * k)$$

Appealing to Boole's inequality again, we may multiply this by the number of epochs and the number of stakeholders to give a bound for the likelihood of generating an invalid chain.

Figure 23 gives the bound on the likelihood of threshold violation for t in our plausible range: from this we can see that the likelihood decreases to a negligible level around 0.21, and so we choose the value of $t = 0.22$, giving an upper bound on the likelihood around $6e - 10$. Increasing t beyond this point gives no decrease in the likelihood of violation.

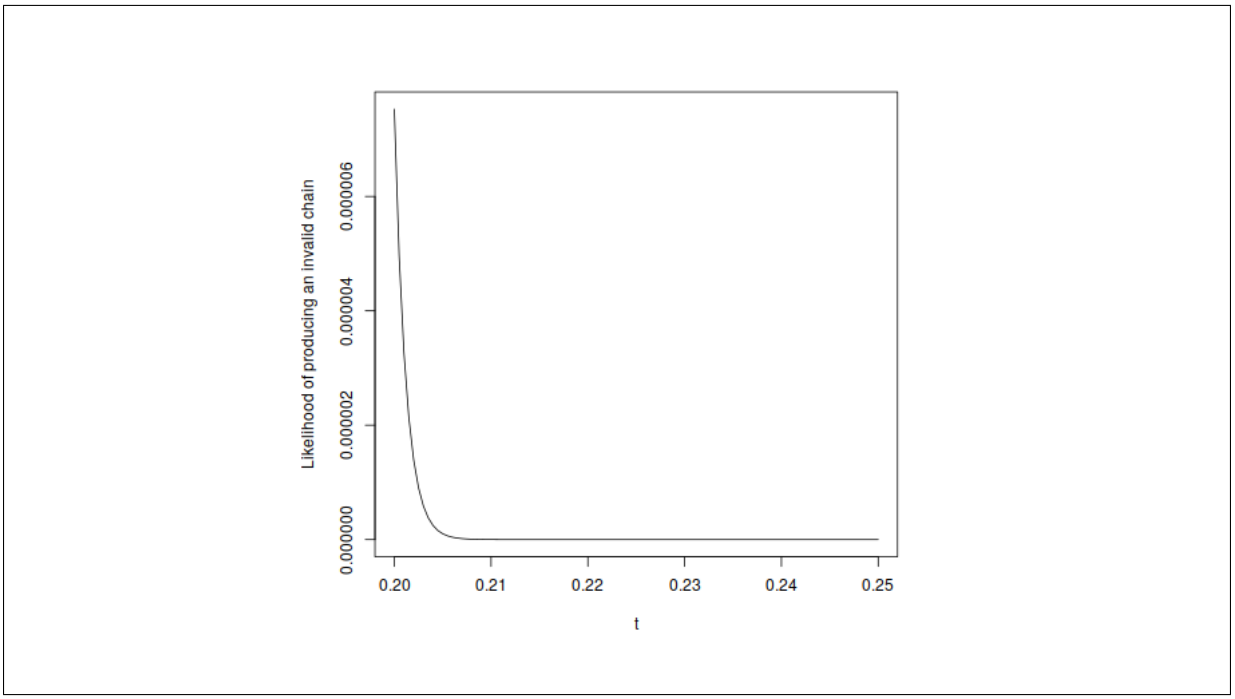

Figure 23: Probability of generating an invalid chain for values of $t \in \left[\frac{1}{4}, \frac{1}{5}\right]$.