Specification of the Blockchain Layer

Marko Dimjašević
Nicholas Clarke

May 2, 2019

Abstract

This document 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

1 Introduction 2
2 Preliminaries 2
  2.1 Sets 3
3 Update interface 3
4 Permissive BFT 5
  4.1 Counting signed blocks 5
  4.2 Permissive BFT Header Processing 7
5 Epoch transitions 9
6 Block processing 11
  6.1 Block header processing 11
  6.2 Block body processing 13
7 Blockchain Extension 15
Appendices 17
A Calculating the $t$ parameter 17

List of Figures

1 Definition of the Union Override Operation 3
2 Update interface processing types and functions 3
3 Update interface processing transition-system types 4
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 Nadalès (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, \( \mathbb{P} X \) 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 \( \epsilon \), and given a sequence \( \Lambda \), \( \Lambda \cdot x \) is the sequence that results from appending \( x \in X \) to \( \Lambda \). Furthermore, \( \epsilon \) is an identity element for sequence joining: \( \epsilon \cdot x = x \cdot \epsilon = x \).

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

Appending with a moving window Given a sequence \( \Lambda \), we define

\[
\Lambda \downarrow \omega \ x := (\Lambda \cdot x) \downarrow (|\Lambda| + 1 - \omega)
\]

Filtering on sequences Given a sequence \( \Lambda \), and a predicate \( p \) on the elements of \( \Lambda \), filter \( p \ \Lambda \) is the sequence that contains all the elements of \( \Lambda \) that satisfy \( p \), in the same order they appear on \( \Lambda \).
\[ K \triangleleft M = \{ i \mapsto o \mid i \mapsto o \in M, i \in K \} \] domain restriction

\[ K \nless M = \{ i \mapsto o \mid i \mapsto o \in M, i \notin K \} \] domain exclusion

\[ M \triangleright V = \{ i \mapsto o \mid i \mapsto o \in M, o \in V \} \] range restriction

\[ \bigcup (A \mapsto B) \rightarrow (A \mapsto B) \rightarrow (A \mapsto B) \] union override

\[ d_0 \subseteq d_1 = d_1 \cup (\text{dom } d_1 \nless d_0) \]

**Figure 1:** Definition of the Union Override Operation

**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 \( \text{patt} = \text{exp} \) to pattern-match an expression \( \text{exp} \) with a certain pattern \( \text{patt} \). For instance, we use \( \Lambda'; x = \Lambda \) to be able to deconstruct a sequence \( \Lambda \) 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 \( \mathbb{N} = \{ 0, 1, 2, \ldots \} \).

**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**

\[
\text{UpdatePayload} = \left( \begin{array}{l}
mprop \in \text{UProp}^? \quad \text{possible update proposal} \\
votes \in \text{Vote}^* \quad \text{votes for update proposals} \\
end \in (\text{VKey} \times \text{ProtVer}) \quad \text{protocol version endorsement}
\end{array} \right)
\]

**Figure 2:** Update interface processing types and functions
Update interface processing transitions

\[
\vdash \begin{array}{c}
\bullet \\
\begin{array}{c}
\rightarrow \\
\text{BUPI} \\
\subseteq \\
\Pi
\end{array}
\end{array}
(\text{UPIEnv} \times \text{UPIState} \times \text{UpdatePayload} \times \text{UPIState})
\]

Figure 3: Update interface processing transition-system types

\[
\Gamma \vdash \begin{array}{c}
\text{us} \xrightarrow{\text{prop}} \text{us}' \\
\text{us}' \xrightarrow{\text{votes}} \text{us}'' \\
\text{us}'' \xrightarrow{\text{end}} \text{us}'''
\end{array}
\]

\[
\Gamma \vdash \begin{array}{c}
\text{us} \xrightarrow{\text{prop}} \text{us}' \\
\text{us} \xrightarrow{\text{end}} \text{us}''
\end{array}
\]

\[
\Gamma \vdash \begin{array}{c}
\text{us} \xrightarrow{\text{votes}} \text{us}' \\
\text{us} \xrightarrow{\text{votes}} \text{us}'
\end{array}
\]

\[
\Gamma \vdash \begin{array}{c}
\text{us} \xrightarrow{\text{end}} \text{us}'' \\
\text{us} \xrightarrow{\text{end}} \text{us}''
\end{array}
\]

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 476th 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.

<table>
<thead>
<tr>
<th>Abstract functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k \in \mathbb{N} )</td>
</tr>
<tr>
<td>( t \in \left[\frac{1}{5}, \frac{1}{4}\right] )</td>
</tr>
<tr>
<td>( dms \in \text{DISTate} \rightarrow (\text{VKey}_G \leftrightarrow \text{VKey}) )</td>
</tr>
</tbody>
</table>

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

\[ \text{BSCEnv} = (\ ds \in \text{DIState} \text{ delegation state} ) \]

Block signature count transitions

\[ \vdash - \xrightarrow{\text{SIGCNT}} - \subseteq \mathcal{P} (\text{BSCEnv} \times \text{VKey}_G^* \times \text{VKey} \times \text{VKey}_G^*) \]

Figure 6: Block signature count transition-system types

\[
\begin{align*}
\{ \text{vk}_g \} &:= \text{dom} \ ((\text{dms} \ ds) \triangleright \{ \text{vk}_d \}) \\
\text{sgs}' &:= \text{sgs}; \text{vk}_g \\
\text{filter} (\text{=} \text{vk}_g) \ | \text{sgs}' | &\leq k \cdot t \\
\text{ds} &\vdash \text{sgs} \xrightarrow{\text{SIGCNT}} \text{sgs}'
\end{align*}
\]

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 verify that the block signer is the current delegate of at least one of the authorised genesis keys.

4. We check that the previous block hash contained in the block header corresponds to the known hash of the previous block.

5. We check that the header signature correctly verifies the “signed” content of the header.

6. Finally, we verify and update the state according to the signature count rules.

Abstract types

\[ bh \in \text{BlockHeader} \quad \text{Block header} \]

Abstract functions

\[ \begin{align*}
\text{bhPrevHash} & \in \text{BlockHeader} \to \text{Hash} \quad \text{previous header hash} \\
\text{bhHash} & \in \text{BlockHeader} \to \text{Hash} \quad \text{header hash} \\
\text{bhSig} & \in \text{BlockHeader} \to \text{Sig} \quad \text{block signature} \\
\text{bhIssuer} & \in \text{BlockHeader} \to \text{VKey} \quad \text{block issuer} \\
\text{bhSlot} & \in \text{BlockHeader} \to \text{Slot} \quad \text{slot for which this block is issued}
\end{align*} \]

Figure 8: Permissive BFT types and functions

Permissive BFT environments

\[ \text{PBFTEnv} = \left( \begin{array}{c}
\text{ds} \in \text{DIState} \\
\text{s}_{\text{last}} \in \text{Slot} \\
\text{s}_{\text{now}} \in \text{Slot}
\end{array} \right) \quad \text{delegation state} \]

\[ \text{PBFTEnv} = \left( \begin{array}{c}
\text{ds} \in \text{DIState} \\
\text{s}_{\text{last}} \in \text{Slot} \\
\text{s}_{\text{now}} \in \text{Slot}
\end{array} \right) \quad \text{delegation state} \]

Permissive BFT states

\[ \text{PBFTState} = \left( \begin{array}{c}
\text{h} \in \text{Hash} \\
\text{sgs} \in \text{VKey}_G^+ \\
\text{Last signers}
\end{array} \right) \quad \text{Tip header hash} \]

Permissive BFT transitions

\[ \text{PBFT} \vdash \text{PBFT} \subseteq \mathbb{P} \quad \text{PBFTEnv \times PBFTState \times BlockHeader \times PBFTState} \]

Figure 9: Permissive BFT transition-system types
\( \text{Figure 10: Permissive BFT rules} \)
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 \( s_{\text{Epoch}} \), 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.

<table>
<thead>
<tr>
<th>Abstract functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_{\text{Epoch}} \in \text{Slot} \rightarrow \text{Epoch} ) epoch containing this slot</td>
</tr>
<tr>
<td>( ngk \in \mathbb{N} ) number of genesis keys</td>
</tr>
</tbody>
</table>

**Figure 11: Epoch transition types and functions**

<table>
<thead>
<tr>
<th>Epoch transition environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ETEnv} = (dms \in \text{VKey}_G \mapsto \text{VKey} \text{ delegation map} \quad e_c \in \text{Epoch} \text{ Current epoch}) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Epoch transition states</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ETState} = (us \in \text{UPIState} \text{ update interface state}) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Epoch transition transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( - \vdash - \rightarrow_{\text{EPOCH}} - \subseteq \mathbb{P} (\text{ETEnv} \times \text{ETState} \times \text{Slot} \times \text{ETState}) )</td>
</tr>
</tbody>
</table>

**Figure 12: Epoch transition transition-system types**
\[
\frac{e_c \geq \text{sEpoch } s}{(dms)_{e_c} \vdash (us) \xrightarrow{s \text{ EPOCH}} (us)}
\]

\[
\frac{e_c < \text{sEpoch } s \quad s \vdash us \xrightarrow{\text{UPIEC}} us'}{(dms)_{e_c} \vdash (us) \xrightarrow{s \text{ EPOCH}} (us')}
\]

**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.

<table>
<thead>
<tr>
<th>Abstract types</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b \in \text{Block} )</td>
</tr>
<tr>
<td>( h \in \text{Hash} )</td>
</tr>
<tr>
<td>( data \in \text{Data} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{bSize} \in \text{Block} \rightarrow \mathbb{N} )</td>
</tr>
<tr>
<td>( \text{verify} \in \text{VKey} \times \text{Data} \times \text{Sig} )</td>
</tr>
</tbody>
</table>

**Figure 14: Basic Block-related Types and Functions**

6.1 Block header processing

Figure 16 gives the transition system types for block header processing. The maxHeaderSize protocol parameter is defined in Damian Nadales (2019). The header state contains the current update state and historical epoch length map, which must be updated along the epoch boundary. Figure 17 gives the corresponding transition rules. During header processing we verify the following things:

1. We compute whether this block is the first block in a new epoch. If so, we apply the epoch transition updates. Unlike other cases, we use the resulting update state in all downstream verification for this block, since the epoch transition is not contingent on the block itself, but simply a recognition that this is a block in a new epoch.

2. We verify that the block header does not exceed the maximum size specified in the protocol parameters.

<table>
<thead>
<tr>
<th>Abstract types</th>
</tr>
</thead>
<tbody>
<tr>
<td>( bh \in \text{BlockHeader} )</td>
</tr>
<tr>
<td>( bts \in \text{BHToSign} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{bHead} \in \text{Block} \rightarrow \text{BlockHeader} )</td>
</tr>
<tr>
<td>( \text{bHeaderSize} \in \text{BlockHeader} \rightarrow \mathbb{N} )</td>
</tr>
</tbody>
</table>

**Figure 15: Block header processing types and functions**
Block header processing environments

\[ \text{BHEnv} = \left( \begin{array}{c}
\text{dms} \in \text{VKey}_c \mapsto \text{VKey} \\
\text{s}_{\text{last}} \in \text{Slot}
\end{array} \right) \text{ delegation map}
\begin{array}{c}
\text{slot of the last seen block}
\end{array} \]

Block header processing states

\[ \text{BHState} = \left( \begin{array}{c}
\text{us} \in \text{UPIState} \\
\text{update state}
\end{array} \right) \]

Block header processing transitions

\begin{align*}
\vdash - \xrightarrow{\text{BHEnv}} - \subseteq \mathcal{P} (\text{BHEnv} \times \text{BHState} \times \text{BlockHeader} \times \text{BHState})
\end{align*}

Figure 16: Block header processing transition-system types

\begin{align*}
\begin{array}{c}
\left( \begin{array}{c}
\text{dms} \\
\text{sEpoch} \\
\text{s}_{\text{last}}
\end{array} \right) \vdash \left( \begin{array}{c}
\text{us} \\
\text{bSlot}
\end{array} \right) \xrightarrow{\text{EPOCH}} \left( \begin{array}{c}
\text{us}'
\end{array} \right) \\
\text{maxHeaderSize} \mapsto \text{s}_{\text{max}} \in \text{pps} \\
\text{bHeaderSize} \leq \text{s}_{\text{max}}
\end{array} \end{align*}

\begin{align*}
\left( \begin{array}{c}
\text{dms} \\
\text{s}_{\text{last}}
\end{array} \right) \vdash \left( \begin{array}{c}
\text{us}
\end{array} \right) \xrightarrow{\text{BHEAD}} \left( \begin{array}{c}
\text{us}'
\end{array} \right)
\end{align*}

Figure 17: Block header processing rules
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 20, where the types and the functions used there are defined in Figure 19. 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

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

Derived functions

\[
\begin{align*}
\text{bEndorsment} & \in \text{Block} \to \text{ProtVer} \times \text{VKey} & \text{Protocol version endorsement} \\
\text{bEndorsment} b & = (\text{bProtVer} b, (\text{bhIssuer} \cdot \text{bHead}) b) \\
\text{bSlot} & \in \text{Block} \to \text{Slot} & \text{Slot for which this block is being issued} \\
\text{bSlot} b & = (\text{bhSlot} \cdot \text{bHead}) b \\
\text{bUpdPayload} & \in \text{Block} \to (\text{UProp}^? \times \text{Vote}^*) & \text{Block update payload} \\
\text{bUpdPayload} b & = (\text{bUpdProp} b, \text{bUpdVotes} b)
\end{align*}
\]

Figure 18: Block body processing types and functions
Block body processing environments

\[
\text{BBEnv} = \left( \begin{array}{c}
\text{pps} \in \text{PParams} \\
\text{e}_n \in \text{Epoch} \\
\text{utxo}_0 \in \text{UTxO}
\end{array} \right)
\begin{array}{c}
\text{protocol parameters} \\
\text{epoch we are currently processing blocks for} \\
\text{genesis UTxO}
\end{array}
\]

Block body processing states

\[
\text{BBState} = \left( \begin{array}{c}
\text{utxoSt} \in \text{UTxOState} \\
\text{ds} \in \text{DIState} \\
\text{us} \in \text{UPIState}
\end{array} \right)
\begin{array}{c}
\text{UTxO state} \\
\text{delegation state} \\
\text{update interface state}
\end{array}
\]

Block body processing transitions

\[
\vdash \quad \text{\text{BBODY}} \quad \subseteq \quad \text{IP (BBEnv} \times \text{BBState} \times \text{Block} \times \text{BBState})
\]

**Figure 19:** Block body processing transition-system types

\[
\text{maxBlockSize} \mapsto b_{\text{max}} \in \text{pps} \\
\text{bSize} b \leq b_{\text{max}} \\
\text{bh} := \text{bHead } b \\
\text{vk}_d := \text{bhIssuer } bh \\
\text{hash (bUtxo } b) = bhUtxoHash \text{bh} \\
\text{hash (bCerts } b) = bhDlhHash \text{bh} \\
\text{hash (bUpdPayload } b) = bhUpdHash \text{bh}
\]

\[
\left( \begin{array}{c}
\text{bSlot } b \\
\text{dom } (\text{dms } ds)
\end{array} \right) \vdash \text{us} \quad \frac{\left( \begin{array}{c}
\text{bUpdProp } b \\
\text{bUpdVotes } b \\
\text{bEndorsment } b
\end{array} \right)}{\text{BUPI}} \rightarrow \text{us}'
\]

\[
\left( \begin{array}{c}
\text{e}_n \\
\text{bSlot } b
\end{array} \right) \vdash \text{ds} \quad \frac{\text{bCerts } b}{\text{DELEG}} \rightarrow \text{ds}' \\
\left( \begin{array}{c}
\text{utxo}_0 \\
\text{pps}
\end{array} \right) \vdash \text{utxoSt} \quad \frac{\text{utxo}_0}{\text{UTXOWS}} \rightarrow \text{utxoSt'}
\]

**Figure 20:** Block body processing rules
7 Blockchain Extension

Figure 23 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 both the header and body according to the rules in figures 17 and 20 respectively.

Abstract functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Domain</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blsEBB</td>
<td>Block</td>
<td>Bool</td>
<td>epoch boundary block check</td>
</tr>
<tr>
<td>pps</td>
<td>UPIState</td>
<td>PParams</td>
<td>update-state protocol-parameters</td>
</tr>
</tbody>
</table>

**Figure 21:** Blockchain Extension Types and Functions

Chain extension environments

\[ \text{CEEnv} = (s_{\text{now}} \in \text{Slot}, \text{current slot}) \]

Chain extension states

\[ \text{CEState} = (s_{\text{last}} \in \text{Slot}, \text{slot of the last seen block}) \]

\[ (sgs \in \text{VKey}_G^+, \text{last signers}) \]

\[ (h \in \text{Hash}, \text{current block hash}) \]

\[ (\text{utxoSt} \in \text{UTxO}, \text{UTxOState}) \]

\[ (ds \in \text{DIState}, \text{delegation state}) \]

\[ (us \in \text{UPIState}, \text{update interface state}) \]

Chain extension transitions

\[ \vdash \xrightarrow{\text{CHAIN}} \subseteq \mathcal{P}(\text{CEEnv} \times \text{CEState} \times \text{Block} \times \text{CEState}) \]

**Figure 22:** Blockchain extension transition-system types
Figure 23: Blockchain extension rules
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 compromised, 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 $X \sim B(k, \frac{1}{7})$.

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

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

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 24 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 24: Probability of generating an invalid chain for values of $t \in \left[\frac{1}{3}, 5\right]$. 