A Formal Specification of the Cardano Ledger

(for the Byron release)

Damian Nadales
damian.nadales@iohk.io

January 4, 2023

Abstract

This document defines the rules for extending a ledger with transactions, as implemented in the Byron release of the Cardano Ledger. It is intended to serve as the specification for random generators of transactions which adhere to the rules presented here.

List of Contributors

Jared Corduan, Nicholas Clarke, Marko Dimjašević, Duncan Coutts, Ru Horlick, Michael Hueschen, Ryan Lemmer.

Contents

List of Figures

1 Introduction

This specification models the *conditions* that the different parts of a transaction have to fulfill so that they can extend a ledger, which is represented here as a list of transactions. In particular, we model the following aspects:

Preservation of value relationship between the total value of input and outputs in a new transaction, and the unspent outputs.

Witnesses authentication of parts of the transaction data by means of cryptographic entities (such as signatures and private keys) contained in these transactions.

Delegation validity of delegation certificates, which delegate block-signing rights.

Update validation voting mechanism which captures the identification of the voters, and the participants that can post update proposals.

The following aspects will not be modeled (since they are not part of the Byron release):

Stake staking rights associated to an addresses.

2 Notation

Natural Numbers The set $\mathbb{N}$ refers to the set of all natural numbers $\{0, 1, 2, \ldots\}$. The set $\mathbb{Q}$ refers to the set of rational numbers.

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

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

Sequences Given a set X, X^* is the set of sequences having elements taken from X. The empty sequence is denoted by ϵ, and given a sequence Λ, $\Lambda; \mathsf{x}$ is the sequence that results from appending $\mathsf{x} \in \mathsf{X}$ to Λ.

Functions $A \to B$ denotes a **total function** from A to B. Given a function f we write $f\ a$ for the application of f to argument a.

Inverse Image Given a function $f : A \to B$ and $b \in B$, we write $f^{-1}\ b$ for the **inverse image** of f at b, which is defined by $\{a \mid f a = b\}$.

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 both $m\ a = b$ and a $m = b$. Given a set A, $A \mapsto A$ represents the identity map on A: $\{a \mapsto a \mid a \in A\}$. The $\varnothing$ symbol is also used to represent the empty map as well.

Domain and range Given a relation $R \in \mathbb{P}\ (A \times B)$, dom $R \in \mathbb{P}\ A$ refers to the domain of R, and range $R \in \mathbb{P}\ B$ refers to the range of R. Note that (partial) functions (and hence maps) are also relations, so we will be using dom and range on functions.

Domain and range operations Given a relation $R \in \mathbb{P}\ (A \times B)$ we make use of the *domain-restriction*, *domain-exclusion*, and *range-restriction* operators, which are defined in Figure 1. Note that a map $A \mapsto B$ can be seen as a relation, which means that these operators can be applied to maps as well.

Integer ranges Given $a, b \in \mathbb{Z}$, $[a, b]$ denotes the sequence $[i \mid a \leq i \leq b]$. Ranges can have open ends: $[.., b]$ denotes sequence $[i \mid i \leq b]$, whereas $[a, ..]$ denotes sequence $[i \mid a \leq i]$. Furthermore, sometimes we use $[a, b]$ to denote a set instead of a sequence. The context in which it is used should provide enough information about the specific type.

Domain and range operations on sequences We overload the $\lhd$, $\not\lhd$, and $\rhd$ to operate over sequences. So for instance given $S \in A^*$, and $R \in (A \times B)^*$: $S \lhd R$ denotes the sequence $[(a, b) \mid (a, b) \in R, a \in S]$.

Wildcard variables When a variable is not needed in a term, we replace it by _ to make it explicit that we do not use this variable in the scope of the given term.

Implicit existential quantifications Given a predicate $P \in X \to \mathbb{B}$, we use $P\_$ as a shorthand notation for $\exists x \cdot P\ x$.

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.

Ceiling Given a number $n \in \mathbb{R}$, $\lceil n \rceil$ represents the ceiling of n, and $\lfloor n \rfloor$ represents its floor.

$$S \lhd R = \{(a,b) \mid (a,b) \in R,\ a \in S\} \qquad \text{domain restriction}$$
$$S \lhd\!\!\!\!- R = \{(a,b) \mid (a,b) \in R,\ a \notin S\} \qquad \text{domain exclusion}$$
$$R \rhd S = \{(a,b) \mid (a,b) \in R,\ b \in S\} \qquad \text{range restriction}$$

Figure 1: Domain and range operations

3 Cryptographic primitives

Figure 2 introduces the cryptographic abstractions used in this document. Note that we define a family of serialization functions $[\![\_]\!]_{\mathsf{A}}$, for all types A for which such serialization function can be defined. When the context is clear, we omit the type suffix, and use simply $[\![\_]\!]$.

Abstract types

$$\begin{array}{rcll} vk & \in & \mathsf{SKey} & \text{signing key} \\ vk & \in & \mathsf{VKey} & \text{verifying key} \\ hk & \in & \mathsf{KeyHash} & \text{hash of a key} \\ \sigma & \in & \mathsf{Sig} & \text{signature} \\ d & \in & \mathsf{Data} & \text{data} \end{array}$$

Derived types

$$(sk, vk) \in \mathsf{SkVk} \quad \text{signing-verifying key pairs}$$

Abstract functions

$$\begin{array}{rcll} \mathsf{hash} & \in & \mathsf{VKey} \to \mathsf{KeyHash} & \text{hash function} \\ \mathsf{verify} & \in & \mathsf{VKey} \times \mathsf{Data} \times \mathsf{Sig} & \text{verification relation} \\ [\![\_]\!]_{\mathsf{A}} & \in & \mathsf{A} \to \mathsf{Data} & \text{serialization function for values of type A} \\ \mathsf{sign} & \in & \mathsf{SKey} \to \mathsf{Data} \to \mathsf{Sig} & \text{signing function} \end{array}$$

Constraints

$$\forall (sk, vk) \in \mathsf{SkVk},\ m \in \mathsf{Data},\ \sigma \in \mathsf{Sig} \cdot \mathsf{sign}\ sk\ m = \sigma \Rightarrow \mathsf{verify}\ vk\ m\ \sigma$$

Notation

$$\mathcal{V}^{\sigma}_{vk}\ d = \mathsf{verify}\ vk\ d\ \sigma \qquad \text{shorthand notation for verify}$$

Figure 2: Cryptographic definitions

3.1 A note on serialization

Definition 1 (Distributivity over serialization functions) *For all types* A *and* B*, given a function* $\mathsf{f} \in \mathsf{A} \to \mathsf{B}$*, we say that the serialization function for values of type* A*, namely* $[\![]\!]_{\mathsf{A}}$ *distributes over* f *if there exists a function* $\mathsf{f}_{[\![]\!]}$ *such that for all* $a \in \mathsf{A}$*:*

$$[\![\mathsf{f}\ a]\!]_{\mathsf{B}} = \mathsf{f}_{[\![]\!]}\ [\![a]\!]_{\mathsf{A}} \tag{1}$$

The equality defined in Equation (1) means that the following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{\ \mathsf{f}\ } & B \\ \downarrow{\scriptstyle [\![]\!]_{\mathsf{A}}} & & \downarrow{\scriptstyle [\![]\!]_{\mathsf{B}}} \\ \mathsf{Data} & \xrightarrow{\ \mathsf{f}_{[\![]\!]}\ } & \mathsf{Data} \end{array}$$

Throughout this specification, whenever we use $[\![\mathsf{f}\ a]\!]_{\mathsf{B}}$, for some type B and function $\mathsf{f} \in \mathsf{A} \to \mathsf{B}$, we assume that $[\![]\!]_{\mathsf{A}}$ distributes over f (see for example Rule 6). This property is what allow us to extract a component of the serialized data (if it is available) without deserializing it in the cases in which the deserialization function ($[\![\_]\!]^{-1}_{\mathsf{A}}$) doesn't behave as an inverse of serialization:

$$[\![\_]\!]_A^{-1} \cdot [\![\_]\!]_A \neq \mathsf{id}_A$$

For the cases in which such an inverse exists, given a function f, we can readily define $f_{[\![]\!]}$ as:

$$f_{[\![]\!]} \doteq [\![\_]\!]_B . f . [\![\_]\!]_A^{-1}$$

4 UTxO

The transition rules for unspent transaction outputs are presented in Figure 6. The states of the UTxO transition system, along with their types are defined in Figure 3:

- we define the protocol parameters as an abstract type, this type is made concrete in Section 6, where the update mechanism is discussed.
- The lovelace supply cap (lovelaceCap) is treated as an abstract function in this specification. In the actual system this value equals

$$45 \times 10^{15}$$

Functions on the types introduced in Figure 6 are defined in Figure 4. In particular, note that in function minfee we make use of the fact that the Lovelace type is an alias for the set of the integers numbers ($\mathbb{Z}$).

Rule 3, models the fact that the **reserves** of the system are set to:

$$\mathsf{lovelaceCap} - \mathsf{balance}\ utxo_0$$

The Lovelace amount 45×10^{15} is the initial money supply in the system.

Abstract types

$$\begin{array}{rcll} tx & \in & \mathsf{Tx} & \text{transaction} \\ txid & \in & \mathsf{TxId} & \text{transaction id} \\ ix & \in & \mathsf{Ix} & \text{transaction index} \\ addr & \in & \mathsf{Addr} & \text{address} \\ pps & \in & \mathsf{PParams} & \text{protocol parameters} \end{array}$$

Derived types

$$\begin{array}{rcllcrl} \ell & \in & \mathsf{Lovelace} & = & n & \in & \mathbb{Z} & \text{currency value} \\ txin & \in & \mathsf{TxIn} & = & (txid, ix) & \in & \mathsf{TxId} \times \mathsf{Ix} & \text{transaction input} \\ txout & \in & \mathsf{TxOut} & = & (addr, c) & \in & \mathsf{Addr} \times \mathsf{Lovelace} & \text{transaction output} \\ utxo & \in & \mathsf{UTxO} & = & m & \in & \mathsf{TxIn} \mapsto \mathsf{TxOut} & \text{unspent tx outputs} \end{array}$$

Abstract Functions

$$\begin{array}{rcll} \mathsf{txid} & \in & \mathsf{Tx} \to \mathsf{TxId} & \text{compute transaction id} \\ \mathsf{txbody} & \in & \mathsf{Tx} \to \mathbb{P}\ \mathsf{TxIn} \times (\mathsf{Ix} \mapsto \mathsf{TxOut}) & \text{transaction body} \\ \mathsf{a} & \in & \mathsf{PParams} \to \mathbb{Z} & \text{minimum fee constant} \\ \mathsf{b} & \in & \mathsf{PParams} \to \mathbb{Z} & \text{minimum fee factor} \\ \mathsf{txSize} & \in & \mathsf{Tx} \to \mathbb{Z} & \text{abstract size of a transaction} \\ \mathsf{lovelaceCap} & \in & \mathsf{Lovelace} & \text{lovelace supply cap} \end{array}$$

Constraints

$$\forall tx_i, tx_j \cdot \mathsf{txid}\ tx_i = \mathsf{txid}\ tx_j \Rightarrow tx_i = tx_j \tag{2}$$

Figure 3: Definitions used in the UTxO transition system

Rule 4 specifies under which conditions a transaction can be applied to a set of unspent outputs, and how the set of unspent output changes with a transaction:

- Each input spent in the transaction must be in the set of unspent outputs.

$\mathsf{txins} \in \mathsf{Tx} \to \mathbb{P}\ \mathsf{TxIn}$ — transaction inputs

$\mathsf{txins}\ tx = inputs \quad \textbf{where} \quad \mathsf{txbody}\ tx = (inputs,\ \_)$

$\mathsf{txouts} \in \mathsf{Tx} \to \mathsf{UTxO}$ — transaction outputs as UTxO

$$\mathsf{txouts}\ tx = \left\{ (\mathsf{txid}\ tx, ix) \mapsto txout \;\middle|\; \begin{array}{l} (\_, outputs) = \mathsf{txbody}\ tx \\ ix \mapsto txout \ \in outputs \end{array} \right\}$$

$\mathsf{balance} \in \mathsf{UTxO} \to \mathbb{Z}$ — UTxO balance

$$\mathsf{balance}\ utxo = \sum_{(\ \_ \mapsto (\_,\ c)) \in utxo} c$$

$\mathsf{minfee} \in \mathsf{PParams} \to \mathsf{Tx} \to \mathbb{Z}$ — minimum fee

$\mathsf{minfee}\ pps\ tx = \mathsf{a}\ pps + \mathsf{b}\ pps * \mathsf{txSize}\ tx$

Figure 4: Functions used in UTxO rules

UTxO environments

$$\mathsf{UTxOEnv} = \left(\begin{array}{rcll} utxo_0 & \in & \mathsf{UTxO} & \text{genesis UTxO} \\ pps & \in & \mathsf{PParams} & \text{protocol parameters map} \end{array} \right)$$

UTxO states

$$\mathsf{UTxOState} = \left(\begin{array}{rcll} utxo & \in & \mathsf{UTxO} & \text{unspent transaction outputs} \\ reserves & \in & \mathsf{Lovelace} & \text{system's reserves} \end{array} \right)$$

UTxO transitions

$$\_ \vdash \_ \xrightarrow[\text{UTXO}]{} \_ \subseteq \mathbb{P}\ (\mathsf{UTxOEnv} \times \mathsf{UTxOState} \times \mathsf{Tx} \times \mathsf{UTxOState})$$

Figure 5: UTxO transition-system types

$$\frac{}{\left(\begin{array}{l} utxo_0 \\ pps \end{array} \right) \vdash \xrightarrow[\text{UTXO}]{} \left(\begin{array}{l} utxo_0 \\ \mathsf{lovelaceCap} - \mathsf{balance}\ utxo_0 \end{array} \right)} \tag{3}$$

$$\frac{\begin{array}{c} \mathsf{txins}\ tx \subseteq \mathrm{dom}\ utxo \\ fee := \mathsf{balance}\ (\mathsf{txins}\ tx \lhd utxo) - \mathsf{balance}\ (\mathsf{txouts}\ tx) \qquad \mathsf{minfee}\ pps\ tx \leq fee \\ \mathsf{txins}\ tx \neq \emptyset \qquad \mathsf{txouts}\ tx \neq \emptyset \qquad \forall \_ \mapsto (\_, c) \in \mathsf{txouts}\ tx \cdot 0 < c \end{array}}{\left(\begin{array}{l} utxo_0 \\ pps \end{array} \right) \vdash \left(\begin{array}{l} utxo \\ reserves \end{array} \right) \xrightarrow[\text{UTXO}]{tx} \left(\begin{array}{l} (\mathsf{txins}\ tx \lhd\!\!\!\!\!- utxo) \cup \mathsf{txouts}\ tx \\ reserves + fee \end{array} \right)} \tag{4}$$

Figure 6: UTxO inference rules

- The minimum fee, which depends on the transaction and the protocol parameters, must be less or equal than the difference between the balance of the unspent outputs in a transaction (i.e. the total amount paid in a transaction) and the amount of spent inputs.
- The set of inputs must not be empty.
- All the transaction outputs must be positive. We do not allow 0 value outputs to be consistent with the current implementation.
- If the above conditions hold, then the new state will not have the inputs spent in transaction tx and it will have the new outputs in tx.

4.1 Witnesses

The rules for witnesses are presented in Figure 10. In the initial rules note that *utxoEnv* and *utxoSt* are tuples, where $utxoEnv \in \mathsf{UTxOEnv}$ and $utxoSt \in \mathsf{UTxOState}$. The definitions used in Rule 6 are given in Figure 7. Note that Rule 6 uses the transition relation defined in Figure 6. The main reason for doing this is to define the rules incrementally, modeling different aspects in isolation to keep the rules as simple as possible. Also note that the $\xrightarrow[\text{UTXO}]{}$ relation could have been defined in terms of $\xrightarrow[\text{UTXOW}]{}$ (thus composing the rules in a different order). The choice here is arbitrary.

Abstract functions

$$\begin{array}{rcll} \mathsf{wits} & \in & \mathsf{Tx} \to \mathbb{P}\,(\mathsf{VKey} \times \mathsf{Sig}) & \text{witnesses of a transaction} \\ \mathsf{hash}_{\mathsf{vkey}} & \in & \mathsf{Addr} \mapsto \mathsf{KeyHash} & \text{hash of a verifying key in an address} \end{array}$$

Figure 7: Definitions used in the UTxO transition system with witnesses

$$\mathsf{addr} \in \mathsf{UTxO} \to \mathsf{TxIn} \mapsto \mathsf{Addr} \qquad \text{addresses of inputs}$$
$$\mathsf{addr}\ utxo = \{i \mapsto a \mid i \mapsto (a, \_) \in utxo\}$$

$$\mathsf{addr}_{\mathsf{h}} \in \mathsf{UTxO} \to \mathsf{TxIn} \mapsto \mathsf{KeyHash} \qquad \text{verifying key hashes}$$
$$\mathsf{addr}_{\mathsf{h}}\ utxo = \{i \mapsto h \mid i \mapsto (a, \_) \in utxo \wedge a \mapsto h \in \mathsf{hash}_{\mathsf{vkey}}\}$$

Figure 8: Functions used in rules witnesses

UTxO with witness transitions

$$\_ \vdash \_ \xrightarrow[\text{UTXOW}]{} \_ \subseteq \mathbb{P}\,(\mathsf{UTxOEnv} \times \mathsf{UTxOState} \times (\mathsf{Tx} \times \mathbb{P}\,(\mathsf{VKey} \times \mathsf{Sig})) \times \mathsf{UTxOState})$$

Figure 9: UTxO with witness transition-system types

$$\frac{utxoEnv \vdash \xrightarrow[\text{UTXO}]{} (utxoSt)}{utxoEnv \vdash \xrightarrow[\text{UTXOW}]{} (utxoSt)} \tag{5}$$

$$\frac{\begin{array}{c} utxoEnv \vdash \begin{pmatrix} utxo \\ reserves \end{pmatrix} \xrightarrow[\text{UTXO}]{tx} \begin{pmatrix} utxo' \\ reserves' \end{pmatrix} \qquad maxTxSize \mapsto t \in pps \qquad \mathsf{txSize}\ tx \leq t \\ \forall i \in \mathsf{txins}\ tx \cdot \exists (vk, \sigma) \in \mathsf{wits}\ tx \cdot \mathcal{V}^{\sigma}_{vk}\,[\![\mathsf{txbody}\ tx]\!] \wedge \mathsf{addr}_{\mathsf{h}}\ utxo\ i = \mathsf{hash}\ vk \end{array}}{utxoEnv \vdash \begin{pmatrix} utxo \\ reserves \end{pmatrix} \xrightarrow[\text{UTXOW}]{tx} \begin{pmatrix} utxo' \\ reserves' \end{pmatrix}} \tag{6}$$

Figure 10: UTxO with witnesses inference rules

4.2 Transaction sequences

Figure 11 models the application of a sequence of transactions. For the sake of concision we omit the types of this transition system, since they are the same as the ones of $\xrightarrow[\text{UTXOW}]{}$.

$$\frac{utxoEnv \vdash \xrightarrow[\text{UTXOW}]{} (utxoSt)}{utxoEnv \vdash \xrightarrow[\text{UTXOWS}]{} (utxoSt)} \tag{7}$$

$$\frac{}{utxoEnv \vdash utxoSt \xrightarrow[\text{UTXOWS}]{\epsilon} utxoSt} \tag{8}$$

$$\frac{utxoEnv \vdash utxoSt \xrightarrow[\text{UTXOWS}]{\Gamma} utxoSt' \qquad utxoEnv \vdash utxoSt' \xrightarrow[\text{UTXOW}]{tx} utxoSt''}{utxoEnv \vdash utxoSt \xrightarrow[\text{UTXOWS}]{\Gamma;tx} utxoSt''} \tag{9}$$

Figure 11: UTxO sequence rules

5 Delegation

An agent owning a key that can sign new blocks can delegate its signing rights to another key by means of *delegation certificates*. These certificates are included in the ledger, and therefore also included in the body of the blocks in the blockchain.

There are several restrictions on a certificate posted on the blockchain:

1. Only genesis keys can delegate.
2. Certificates must be properly signed by the delegator.
3. Any given key can delegate at most once per-epoch.
4. Any given key can issue at most one certificate in a given slot.
5. The epochs in the certificates must refer to the current or to the next epoch. We do not want to allow certificates from past epochs so that a delegation certificate cannot be replayed. On the other hand if we allow certificates with arbitrary future epochs, then a malicious key can issue a delegation certificate per-slot, setting the epoch to a sufficiently large value. This will cause a blow up in the size of the ledger state since we will not be able to clean *eks* (we only clean past epochs). Also note that we do not check the relation between the certificate epoch and the slot in which the certificate becomes active. This would bring additional complexity without any obvious benefit.
6. Certificates do not become active immediately, but they require a certain number of slots till they become stable in all the nodes.

These conditions are formalized in Figure 14. Rule 11 determines when a certificate can become "scheduled". The definitions used in these rules are presented in Figure 12, and the types of the system induced by $\xrightarrow[\text{SDELEG}]{-}$ are presented in Figure 13. Here and in the remaining rules we will be using k as an abstract constant that gives us the chain stability parameter.

Abstract types

$$\begin{array}{rcll} c & \in & \mathsf{DCert} & \text{delegation certificate} \\ vk_g & \in & \mathsf{VKey_G} & \text{genesis verification key} \end{array}$$

Derived types

$$\begin{array}{rcrcll} e & \in & \mathsf{Epoch} & = & n \in \mathbb{N} & \text{epoch} \\ s & \in & \mathsf{Slot} & = & s \in \mathbb{N} & \text{slot} \\ d & \in & \mathsf{SlotCount} & = & s \in \mathbb{N} & \text{slot} \end{array}$$

Constraints

$$\mathsf{VKey_G} \subseteq \mathsf{VKey}$$

Abstract functions

$$\begin{array}{rcll} \mathsf{dbody} & \in & \mathsf{DCert} \to (\mathsf{VKey} \times \mathsf{Epoch}) & \text{body of the delegation certificate} \\ \mathsf{dwit} & \in & \mathsf{DCert} \to (\mathsf{VKey_G} \times \mathsf{Sig}) & \text{witness for the delegation certificate} \\ \mathsf{dwho} & \in & \mathsf{DCert} \mapsto (\mathsf{VKey_G} \times \mathsf{VKey}) & \text{who delegates to whom in the certificate} \\ \mathsf{depoch} & \in & \mathsf{DCert} \mapsto \mathsf{Epoch} & \text{certificate epoch} \\ k & \in & \mathbb{N} & \text{chain stability parameter} \end{array}$$

Figure 12: Delegation scheduling definitions

Delegation scheduling environments

$$\mathsf{DSEnv} = \begin{pmatrix} \mathcal{K} & \in & \mathbb{P}\,\mathsf{VKey_G} & \text{allowed delegators} \\ e & \in & \mathsf{Epoch} & \text{epoch} \\ s & \in & \mathsf{Slot} & \text{slot} \end{pmatrix}$$

Delegation scheduling states

$$\mathsf{DSState} = \begin{pmatrix} sds & \in & (\mathsf{Slot} \times (\mathsf{VKey_G} \times \mathsf{VKey}))^* & \text{scheduled delegations} \\ eks & \in & \mathbb{P}\,(\mathsf{Epoch} \times \mathsf{VKey_G}) & \text{key-epoch delegations} \end{pmatrix}$$

Delegation scheduling transitions

$$\_ \vdash \_ \xrightarrow[\text{SDELEG}]{\_} \_ \subseteq \mathbb{P}\,(\mathsf{DSEnv} \times \mathsf{DSState} \times \mathsf{DCert} \times \mathsf{DSState})$$

Figure 13: Delegation scheduling transition-system types

$$\frac{sds_0 := \epsilon \qquad eks_0 := \emptyset}{\begin{pmatrix} \mathcal{K} \\ e \\ s \end{pmatrix} \vdash \xrightarrow[\text{SDELEG}]{} \begin{pmatrix} sds_0 \\ eks_0 \end{pmatrix}} \tag{10}$$

$$\frac{\begin{array}{ccc} (vk_s,\ \sigma) := \mathsf{dwit}\ c & \mathsf{verify}\ vk_s\ [\![\mathsf{dbody}\ c]\!]\ \sigma & vk_s \in \mathcal{K} \\ (vk_s,\ vk_d) := \mathsf{dwho}\ c \quad e_d := \mathsf{depoch}\ c & (e_d,\ vk_s) \notin eks & 0 \leq e_d - e \leq 1 \\ d := 2 \cdot k & (s+d,\ (vk_s,\ \_)) \notin sds & \end{array}}{\begin{pmatrix} \mathcal{K} \\ e \\ s \end{pmatrix} \vdash \begin{pmatrix} sds \\ eks \end{pmatrix} \xrightarrow[\text{SDELEG}]{c} \begin{pmatrix} sds;(s+d,\ (vk_s,\ vk_d)) \\ eks \cup \{(e_d,\ vk_s)\} \end{pmatrix}} \tag{11}$$

Figure 14: Delegation scheduling rules

The rules in Figure 17 model the activation of delegation certificates. Once a scheduled certificate becomes active (see Section 7.1.1), the delegation map is changed by it only if:

- The delegating key (vk_s) did not activate a delegation certificate in a slot greater or equal than the certificate slot (s). This check is performed to avoid having the constraint that the delegation certificates have to be activated in slot order.
- The key being delegated to (vk_d) has not been delegated by another key (injectivity constraint).

The reason why we check that the delegation map is injective is to avoid a potential risk (during the OBFT era) in which a malicious node gets control of a genesis key vk_m that issued the maximum number of blocks in a given window. By delegating to another key vk_d, which was already delegated to by some other key vk_g, the malicious node could prevent vk_g from issuing blocks. Even though the delegation certificates take several slots to become effective, the malicious node could calculate when the certificate would become active, and issue a delegation certificate at the right time.

As an additional advantage, by having an injective delegation map, we are able to simplify our specification when it comes to counting the blocks issued by (delegates of) genesis keys.

Note also, that we could not impose the injectivity constraint in Rule 11 since we do not have information about the delegations that will become effective. We could of course detect a violation in the injectivity constraint when scheduling a delegation certificate, but this will lead to a complex computation and larger state in said rule.

Finally, note that we do not want to reject a scheduled delegation that would violate the injectivity constraint (since delegation might not have been scheduled by the node issuing the block). Instead, we simply ignore the delegation certificate (Rule 14).

$$\underset{\rightarrow}{\cup} \in (A \mapsto B) \to (A \mapsto B) \to (A \mapsto B) \qquad \text{union override}$$
$$d_0 \underset{\rightarrow}{\cup} d_1 = d_1 \cup (\operatorname{dom} d_1 \mathbin{\not\lhd} d_0)$$

Figure 15: Functions used in delegation rules

Delegation environments

$$\mathsf{DEnv} = \left(\begin{array}{lll} \mathcal{K} & \in & \mathbb{P}\ \mathsf{VKey_G} \end{array} \right. \quad \text{allowed delegators} \quad \left. \right)$$

Delegation states

$$\mathsf{DState} = \left(\begin{array}{lll} dms & \in & \mathsf{VKey_G} \mapsto \mathsf{VKey} \\ dws & \in & \mathsf{VKey_G} \mapsto \mathsf{Slot} \end{array} \right. \quad \begin{array}{r} \text{delegation map} \\ \text{when last delegation occurred} \end{array} \quad \left. \right)$$

Delegation transitions

$$\_ \vdash \_ \xrightarrow[\text{ADELEG}]{} \_ \in \mathbb{P}\ (\mathsf{DEnv} \times \mathsf{DState} \times (\mathsf{Slot} \times (\mathsf{VKey_G} \times \mathsf{VKey})) \times \mathsf{DState})$$

Figure 16: Delegation transition-system types

$$\frac{dms_0 := \{\, k \mapsto k \mid k \in \mathcal{K} \,\} \qquad dws_0 := \{\, k \mapsto 0 \mid k \in \mathcal{K} \,\}}{(\mathcal{K}) \vdash \xrightarrow[\text{ADELEG}]{} \begin{pmatrix} dms_0 \\ dws_0 \end{pmatrix}} \tag{12}$$

$$\frac{vk_d \notin \text{range } dms \qquad (vk_s \mapsto s_p \in dws \Rightarrow s_p < s)}{(\mathcal{K}) \vdash \begin{pmatrix} dms \\ dws \end{pmatrix} \xrightarrow[\text{ADELEG}]{(s,\,(vk_s,\,vk_d))} \begin{pmatrix} dms & \underset{\rightarrow}{\cup} & \{vk_s \mapsto vk_d\} \\ dws & \underset{\rightarrow}{\cup} & \{vk_s \mapsto s\} \end{pmatrix}} \tag{13}$$

$$\frac{vk_d \in \text{range } dms \vee (vk_s \mapsto s_p \in dws \wedge s \leq s_p)}{(\mathcal{K}) \vdash \begin{pmatrix} dms \\ dws \end{pmatrix} \xrightarrow[\text{ADELEG}]{(s,\,(vk_s,\,vk_d))} \begin{pmatrix} dms \\ dws \end{pmatrix}} \tag{14}$$

Figure 17: Delegation inference rules

5.1 Delegation sequences

This section presents the rules that model the effect that sequences of delegations have on the ledger.

$$\frac{delegEnv \vdash \xrightarrow[\text{SDELEG}]{} delegSt}{delegEnv \vdash \xrightarrow[\text{SDELEGS}]{} delegSt} \tag{15}$$

$$\frac{}{delegEnv \vdash delegSt \xrightarrow[\text{SDELEGS}]{\epsilon} delegSt} \tag{16}$$

$$\frac{delegEnv \vdash delegSt \xrightarrow[\text{SDELEGS}]{\Gamma} delegSt' \qquad delegEnv \vdash delegSt' \xrightarrow[\text{SDELEG}]{c} delegSt''}{delegEnv \vdash delegSt \xrightarrow[\text{SDELEGS}]{\Gamma;c} delegSt''} \tag{17}$$

Figure 18: Delegation scheduling sequence rules

5.2 Deviation from the `cardano-sl` implementation

In the `cardano-sl` implementation, the block issuer needs to include a delegation certificate in the block, which witness the fact that a genesis key gave the issuer the rights of issuing blocks on behalf of this genesis key. The reasons why this was implemented in this way in `cardano-sl` are not clear, since the delegation certificates are posted on the chain, so the ledger state contains the information about who delegates to whom. Hence in the current specification we use a heavyweight delegation scheme, i.e. where the certificates are posted on the chain, but an implementation of this rules that aims at being compatible with the implementation in `cardano-sl` has to take the fact that delegation certificates are also present in a block into account.

$$\frac{delegEnv \vdash \xrightarrow[\text{ADELEG}]{} delegSt}{delegEnv \vdash \xrightarrow[\text{ADELEGS}]{} delegSt} \tag{18}$$

$$\frac{}{delegEnv \vdash delegSt \xrightarrow[\text{ADELEGS}]{\epsilon} delegSt} \tag{19}$$

$$\frac{delegEnv \vdash delegSt \xrightarrow[\text{ADELEGS}]{\Gamma} delegSt' \qquad delegEnv \vdash delegSt' \xrightarrow[\text{ADELEG}]{c} delegSt''}{delegEnv \vdash delegSt \xrightarrow[\text{ADELEGS}]{\Gamma;c} delegSt''} \tag{20}$$

Figure 19: Delegations sequence rules

6 Update mechanism

This section formalizes the update mechanism by which the protocol parameters get updated. This formalization is a simplification of the current update mechanism implemented in `cardano-sl`, and partially documented in:

- Updater implementation
- Update mechanism
- Update system consensus rules

The reason for formalizing a simplified version of the current implementation is that research work on blockchain update mechanisms is needed before introducing a more complex update logic. Since this specification is to be implemented in a federated setting, some of the constraints put in place in the current implementation are no longer relevant. Once the research work is ready, this specification can be extended to incorporate the research results.

6.1 Update proposals

The definitions used in the update mechanism rules are presented in Figure 20. A system tag is used to identify the system for which the update is proposed (in practice this would be a string referring to an operating system; e.g. `linux`, `win64`, or `mac32`). The software update metadata (Mdt) is any information required for performing an update such as hashes of software downloads. Note that the fact that the metadata is kept abstract in the specification does not mean that we allow any arbitrary metadata (in the actual implementation this abstract metadata would correspond to `Map SystemTag UpdateData`, were the `SystemTag` corresponds with STag and `UpdateData` contains the software hash for a specific platform).

The set of protocol parameters (Ppm) is assumed to contain the following keys, some of which correspond with fields of the `cardano-sl` `BlockVersionData` structure:

- Maximum block size: *maxBlockSize*
- Maximum transaction size: *maxTxSize*
- Maximum header size: *maxHeaderSize*
- Maximum proposal size: *maxProposalSize*
- Transaction fee policy: *txFeePolicy*
- Script version: *scriptVersion*
- Update adoption threshold: *upAdptThd*. This represents the minimum percentage of the total number of genesis keys that have to endorse a protocol version to be able to become adopted. We use this parameter to determine the confirmation threshold as well. There is no corresponding parameter in the `cardano-sl` protocol parameters, however we do have a soft-fork minimum threshold parameter (`srMinThd` in `bvdSoftforkRule`). When divided by, 1×10^{15}, it determines the minimum portion of the total stake that is needed for the adoption of a new protocol version. On mainnet, this number is set to 6×10^{14}, so the minimum portion becomes 0.6. This number can be multiplied by the total number of genesis keys to obtain how many keys are needed to reach a majority.
- Update proposal time-to-live: *upropTTL*. This would correspond to the number of slots specified by `bvdUpdateImplicit`. In `cardano-sl` the rule was that after `bvdUpdateImplicit` slots, if a proposal did not reach a majority of the votes, then if the proposal has more votes

Abstract types

$$
\begin{array}{rcll}
up & \in & \mathsf{UProp} & \text{update proposal} \\
p & \in & \mathsf{Ppm} & \text{protocol parameter} \\
upd & \in & \mathsf{UpdData} & \text{update data} \\
upa & \in & \mathsf{UpdAttrs} & \text{update attributes} \\
an & \in & \mathsf{ApName} & \text{application name} \\
t & \in & \mathsf{STag} & \text{system tag} \\
m & \in & \mathsf{Mdt} & \text{metadata}
\end{array}
$$

Derived types

$$
\begin{array}{rcrcll}
s_n & \in & \mathsf{Slot} & = & n \in \mathbb{N} & \text{slot number} \\
pv & \in & \mathsf{ProtVer} & = & (maj, min, alt) \in (\mathbb{N}, \mathbb{N}, \mathbb{N}) & \text{protocol version} \\
pps & \in & \mathsf{PParams} & = & pps \in \mathsf{Ppm} \mapsto \mathsf{Value} & \text{protocol parameters map} \\
apv & \in & \mathsf{ApVer} & = & n \in \mathbb{N} & \\
swv & \in & \mathsf{SWVer} & = & (an, av) \in \mathsf{ApName} \times \mathsf{ApVer} & \text{software version} \\
pb & \in & \mathsf{UpSD} & = & \begin{pmatrix} pv \\ pps \\ swv \\ upd \\ upa \end{pmatrix} \in \begin{pmatrix} \mathsf{ProtVer} \\ \mathsf{PParams} \\ \mathsf{SWVer} \\ \mathsf{UpdData} \\ \mathsf{UpdAttrs} \end{pmatrix} & \text{protocol update signed data}
\end{array}
$$

Abstract functions

$$
\begin{array}{rcll}
\mathsf{upIssuer} & \in & \mathsf{UProp} \to \mathsf{VKey} & \text{update proposal issuer (delegate)} \\
\mathsf{upSize} & \in & \mathsf{UProp} \to \mathbb{N} & \text{update proposal size} \\
\mathsf{upPV} & \in & \mathsf{UProp} \to \mathsf{ProtVer} & \text{update proposal protocol version} \\
\mathsf{upId} & \in & \mathsf{UProp} \to \mathsf{UpId} & \text{update proposal id} \\
\mathsf{upParams} & \in & \mathsf{UProp} \to \mathsf{PParams} & \text{proposed parameters update} \\
\mathsf{upSwVer} & \in & \mathsf{UProp} \to \mathsf{SWVer} & \text{software-version update proposal} \\
\mathsf{upSig} & \in & \mathsf{UProp} \to \mathsf{Sig} & \text{update proposal signature} \\
\mathsf{upSigData} & \in & \mathsf{UProp} \to \mathsf{UpSD} & \text{update proposal signed data} \\
\mathsf{upSTags} & \in & \mathsf{UProp} \to \mathbb{P}\ \mathsf{STag} & \text{update proposal system tags} \\
\mathsf{upMdt} & \in & \mathsf{UProp} \to \mathsf{Mdt} & \text{software update metadata}
\end{array}
$$

Figure 20: Update proposals definitions

for than against it, then it will become implicitly accepted, or rejected otherwise. In this specification, we re-interpret the meaning of this parameter as the proposal time-to-live: if after the number of slots specified by `bvdUpdateImplicit` the proposal does not reach a majority of approvals, the proposal is simply discarded. In the mainnet configuration (`mainnet-genesis.json`) this value is set to 10000, which corresponds with almost half of the total number of slots in an epoch.

The protocol parameters are formally defined in Figure 21.

6.2 Update proposals registration

The rules in Figure 25 model the validity of a proposal:

- if an update proposal proposes a change in the protocol version, it must do so in a consistent manner:
  - The proposed version must be lexicographically bigger than the current version.

$$
\begin{array}{rcll}
\mathit{maxBlockSize} \mapsto \mathbb{N} & \in & \mathsf{PParams} & \text{maximum block size} \\
\mathit{maxTxSize} \mapsto \mathbb{N} & \in & \mathsf{PParams} & \text{maximum transaction size} \\
\mathit{maxHeaderSize} \mapsto \mathbb{N} & \in & \mathsf{PParams} & \text{maximum header size} \\
\mathit{scriptVersion} \mapsto \mathbb{N} & \in & \mathsf{PParams} & \text{script version} \\
\mathit{upAdptThd} \mapsto \mathbb{Q} & \in & \mathsf{PParams} & \text{update proposal adoption threshold} \\
\mathit{upropTTL} \mapsto \mathsf{Slot} & \in & \mathsf{PParams} & \text{update proposal time-to-live}
\end{array}
$$

Figure 21: Protocol-parameters definitions

Update proposals validity environments

$$
\mathsf{UPVEnv} = \left(\begin{array}{rcll}
\mathit{pv} & \in & \mathsf{ProtVer} & \text{adopted (current) protocol version} \\
\mathit{pps} & \in & \mathsf{PParams} & \text{adopted protocol parameters map} \\
\mathit{avs} & \in & \mathsf{ApName} \mapsto (\mathsf{ApVer} \times \mathsf{Slot} \times \mathsf{Mdt}) & \text{application versions}
\end{array} \right)
$$

Update proposals validity states

$$
\mathsf{UPVState} = \left(\begin{array}{rcll}
\mathit{rpus} & \in & \mathsf{UpId} \mapsto (\mathsf{ProtVer} \times \mathsf{PParams}) & \text{registered protocol update proposals} \\
\mathit{raus} & \in & \mathsf{UpId} \mapsto (\mathsf{ApName} \times \mathsf{ApVer} \times \mathsf{Mdt}) & \text{registered software update proposals}
\end{array} \right)
$$

Update proposals validity transitions

$$
\_ \vdash \_ \xrightarrow[\mathrm{UPV}]{} \_ \subseteq \mathbb{P}\,(\mathsf{UPVEnv} \times \mathsf{UPVState} \times \mathsf{UProp} \times \mathsf{UPVState})
$$

Figure 22: Update proposals validity transition-system types

  - The major versions of the proposed and current version must differ in at most one.
  - If the proposed major version is equal to the current major version, then the proposed minor version must be incremented by one.
  - If the proposed major version is larger than the current major version, then the proposed minor version must be zero.
  - must be consistent with the current protocol parameters:
    * the proposal size must not exceed the maximum size specified by the current protocol parameters, (note that here we use function application to extract the value of the different protocol parameters, and a rule that uses a value of the map can be applied only if the function -e.g. *pps*- is defined for that value)
    * the proposed new maximum block size should be not greater than twice current maximum block size,
    * the maximum transaction size must be smaller than the maximum block size (this requirement is **crucial** for having every transaction fitting in a block, and
    * the proposed new script version can be incremented by at most 1.
  - must have a unique version among the current active proposals.

- if an update proposal proposes to increase the application version version (*av*) for a given application (*an*), then there should not be an active update proposal that proposes the same update.

Note that the rules in Figure 25 allow for an update that does not propose changes in the protocol version, or does not propose changes to the software version. However the update proposal

must contain a change proposal in any of these two aspects. Also note that we do not allow for updating the protocol parameters without updating the protocol version. If an update in the protocol parameters does not cause a soft-fork we might use the alt version for that purpose.

In Rule 24 we make use of the following abstract functions:

- apNameValid, which checks that the name is an ASCII string 12 characters or less.
- sTagValid, which checks that the name is an ASCII string of 10 characters or less.

Abstract functions

$$\begin{array}{rcll} \mathsf{apNameValid} & \in & \mathsf{ApName} \to \mathbb{B} & \text{validity checking for application name} \\ \mathsf{sTagValid} & \in & \mathsf{STag} \to \mathbb{B} & \text{validity checking for system tag} \end{array}$$

Figure 23: Update proposal validity definitions

$$\begin{array}{rcl} \mathsf{pvCanFollow}\ (mj_p, mi_p, a_p)\ (mj_n, mi_n, a_n) & = & (mj_p, mi_p, a_p) < (mj_n, mi_n, a_n) \\ & \wedge & 0 \leq mj_n - mj_p \leq 1 \\ & \wedge & (mj_p = mj_n \Rightarrow mi_p + 1 = mi_n)) \\ & \wedge & (mj_p + 1 = mj_n \Rightarrow mi_n = 0) \end{array} \tag{21}$$

$$\begin{array}{l} \mathsf{canUpdate}\ pps\ pps' \\ \quad = \quad pps'\ maxBlockSize \leq 2 \cdot pps\ maxBlockSize \\ \quad \wedge \quad pps'\ maxTxSize < pps'\ maxBlockSize \\ \quad \wedge \quad 0 \leq pps'\ scriptVersion - pps\ scriptVersion \leq 1 \end{array} \tag{22}$$

$$\begin{array}{rcl} \mathsf{svCanFollow}\ avs\ (an, av) & = & (an \mapsto (av_c, \_, \_) \in avs \Rightarrow av = av_c + 1) \\ & \wedge & (an \notin \mathrm{dom}\ avs \Rightarrow av = 0 \vee av = 1) \end{array} \tag{23}$$

Figure 24: Update validity functions

$$\frac{\begin{array}{c}(an, av) := \mathsf{upSwVer}\ up \qquad \mathsf{apNameValid}\ an \\ \mathsf{svCanFollow}\ avs\ (an, av) \qquad (an, \_, \_) \notin \text{range}\ raus \\ \forall t \in \mathsf{upSTags}\ up \cdot \mathsf{sTagValid}\ t\end{array}}{\left(\ avs\ \right) \vdash \left(\ raus\ \right) \xrightarrow[\text{UPSVV}]{up} \left(\ raus \underset{\rightarrow}{\cup} \{\mathsf{upId}\ up \mapsto (an, av, \mathsf{upMdt}\ up)\}\ \right)} \tag{24}$$

$$\frac{\begin{array}{c}pps' := pps \underset{\rightarrow}{\cup} \mathsf{upParams}\ up \qquad \mathsf{canUpdate}\ pps\ pps' \\ nv := \mathsf{upPV}\ up \qquad \mathsf{pvCanFollow}\ nv\ pv \\ \mathsf{upSize}\ up \leq pps\ maxProposalSize \qquad nv \notin \text{dom}\ (\text{range}\ rpus)\end{array}}{\left(\begin{array}{c}pv \\ pps\end{array}\right) \vdash \left(\ rpus\ \right) \xrightarrow[\text{UPPVV}]{up} \left(\ rpus \underset{\rightarrow}{\cup} \{\mathsf{upId}\ up \mapsto (nv, pps')\}\ \right)} \tag{25}$$

$$\frac{\left(\begin{array}{c}pv \\ pps\end{array}\right) \vdash \left(\ rpus\ \right) \xrightarrow[\text{UPPVV}]{up} \left(\ rpus'\ \right) \qquad (an, av) := \mathsf{upSwVer}\ up \qquad an \mapsto (av, \_, \_) \in avs}{\left(\begin{array}{c}pv \\ pps \\ avs\end{array}\right) \vdash \left(\begin{array}{c}rpus \\ raus\end{array}\right) \xrightarrow[\text{UPV}]{up} \left(\begin{array}{c}rpus' \\ raus\end{array}\right)} \tag{26}$$

$$\frac{pv = \mathsf{upPV}\ up \qquad \mathsf{upParams}\ up = \emptyset \qquad \left(\ avs\ \right) \vdash \left(\ raus\ \right) \xrightarrow[\text{UPSVV}]{up} \left(\ raus'\ \right)}{\left(\begin{array}{c}pv \\ pps \\ avs\end{array}\right) \vdash \left(\begin{array}{c}rpus \\ raus\end{array}\right) \xrightarrow[\text{UPV}]{up} \left(\begin{array}{c}rpus \\ raus'\end{array}\right)} \tag{27}$$

$$\frac{\left(\begin{array}{c}pv \\ pps\end{array}\right) \vdash \left(\ rpus\ \right) \xrightarrow[\text{UPPVV}]{up} \left(\ rpus'\ \right) \qquad avs\ \vdash \left(\ raus\ \right) \xrightarrow[\text{UPSVV}]{up} \left(\ raus'\ \right)}{\left(\begin{array}{c}pv \\ pps \\ avs\end{array}\right) \vdash \left(\begin{array}{c}rpus \\ raus\end{array}\right) \xrightarrow[\text{UPV}]{up} \left(\begin{array}{c}rpus' \\ raus'\end{array}\right)} \tag{28}$$

Figure 25: Update proposals validity rules

The rule of Figure 27 models the registration of an update proposal:

- We consider the update proposal issuers to be the delegators of the key (vk) that is associated with the proposal under consideration (up).
- We check that the issuer of a proposal was delegated by a genesis key (which are in the domain of dms).
- the update proposal data (see the definition of upSigdata) must be signed by the proposal issuer.

Update proposals registration environments

$$\mathsf{UPREnv} = \begin{pmatrix} pv \in \mathsf{ProtVer} & \text{adopted (current) protocol version} \\ pps \in \mathsf{PParams} & \text{adopted protocol parameters map} \\ avs \in \mathsf{ApName} \mapsto (\mathsf{ApVer} \times \mathsf{Slot} \times \mathsf{Mdt}) & \text{application versions} \\ dms \in \mathsf{VKey_G} \mapsto \mathsf{VKey} & \text{delegation map} \end{pmatrix}$$

Update proposals registration states

$$\mathsf{UPRState} = \begin{pmatrix} rpus \in \mathsf{UpId} \mapsto (\mathsf{ProtVer} \times \mathsf{PParams}) & \text{registered update proposals} \\ raus \in \mathsf{UpId} \mapsto (\mathsf{ApName} \times \mathsf{ApVer} \times \mathsf{Mdt}) & \text{registered software update proposals} \end{pmatrix}$$

Update proposals registration transitions

$$\_ \vdash \_ \xrightarrow[\text{UPREG}]{} \_ \subseteq \mathbb{P}\,(\mathsf{UPREnv} \times \mathsf{UPRState} \times \mathsf{UProp} \times \mathsf{UPRState})$$

Figure 26: Update proposals registration transition-system types

$$\frac{\begin{pmatrix} pv \\ pps \\ avs \end{pmatrix} \vdash \begin{pmatrix} rpus \\ raus \end{pmatrix} \xrightarrow[\text{UPV}]{up} \begin{pmatrix} rpus' \\ raus' \end{pmatrix} \qquad dms \triangleright \{vk\} \neq \emptyset \qquad vk := \mathsf{upIssuer}\ up \qquad \mathcal{V}_{vk}[\![\mathsf{upSigData}\ up]\!]_{(\mathsf{upSig}\ up)}}{\begin{pmatrix} pv \\ pps \\ avs \\ dms \end{pmatrix} \vdash \begin{pmatrix} rpus \\ raus \end{pmatrix} \xrightarrow[\text{UPREG}]{up} \begin{pmatrix} rpus' \\ raus' \end{pmatrix}} \tag{29}$$

Figure 27: Update registration rules

6.3 Voting on update proposals

Abstract types

$$v \in \mathsf{Vote} \quad \text{vote on an update proposal}$$

Abstract functions

$$\begin{array}{ll} \mathsf{vCaster} \in \mathsf{Vote} \to \mathsf{VKey} & \text{caster of a vote} \\ \mathsf{vPropId} \in \mathsf{Vote} \to \mathsf{UpId} & \text{proposal id that is being voted} \\ \mathsf{vSig} \in \mathsf{Vote} \to \mathsf{Sig} & \text{vote signature} \end{array}$$

Figure 28: Voting definitions

Voting environments

$$\mathsf{VREnv} = \left(\begin{array}{lll} \mathit{rups} & \in \mathbb{P}\ \mathsf{UpId} & \text{registered update proposals} \\ \mathit{dms} & \in \mathsf{VKey_G} \mapsto \mathsf{VKey} & \text{delegation map} \end{array} \right)$$

Voting states

$$\mathsf{VRState} = \left(\begin{array}{lll} \mathit{vts} & \in \mathbb{P}\ (\mathsf{UpId} \times \mathsf{VKey_G}) & \text{votes} \end{array} \right)$$

Voting transitions

$$\_ \vdash \_ \xrightarrow[\text{ADDVOTE}]{\_} \_ \in \mathbb{P}\ (\mathsf{VREnv} \times \mathsf{VRState} \times \mathsf{Vote} \times \mathsf{VRState})$$

Figure 29: Voting transition-system types

In Rule 30:

- Only genesis keys can vote on an update proposal, although votes can be cast by delegates of these genesis keys.
- We count one vote per genesis key that delegated to the key that is casting the vote.
- The vote must refer to a registered update proposal.
- The proposal id must be signed by the key that is casting the vote.
- A given genesis key is only allowed to vote for a proposal once. This provision guards against replay attacks, where a third party may replay the vote in multiple blocks.

$$\frac{\begin{array}{ccc} \mathit{pid} := \mathsf{vPropId}\ v & \mathit{vk} := \mathsf{vCaster}\ v & \mathit{pid} \in \mathit{rups} \\ \mathit{vts}_{\mathit{pid}} := \{(\mathit{pid}, \mathit{vk}_s) \mid \mathit{vk}_s \mapsto \mathit{vk} \in \mathit{dms}\} & \mathit{vts}_{\mathit{pid}} \neq \emptyset & \mathit{vts}_{\mathit{pid}} \nsubseteq \mathit{vts} \\ \multicolumn{3}{c}{\mathcal{V}_{\mathit{vk}}[\![\mathit{pid}]\!]_{(\mathsf{vSig}\ v)}} \end{array}}{\left(\begin{array}{l} \mathit{rups} \\ \mathit{dms} \end{array} \right) \vdash \left(\begin{array}{l} \mathit{vts} \end{array} \right) \xrightarrow[\text{ADDVOTE}]{v} \left(\begin{array}{l} \mathit{vts} \cup \mathit{vts}_{\mathit{pid}} \end{array} \right)} \tag{30}$$

Figure 30: Update voting rules

Vote registration environments

$$\mathsf{VEnv} = \begin{pmatrix} s_n & \in & \mathsf{Slot} & \text{current slot number} \\ t & \in & \mathbb{N} & \text{confirmation threshold} \\ rups & \in & \mathbb{P}\ \mathsf{UpId} & \text{registered update proposals} \\ dms & \in & \mathsf{VKey_G} \mapsto \mathsf{VKey} & \text{delegation map} \end{pmatrix}$$

Vote registration states

$$\mathsf{VState} = \begin{pmatrix} cps & \in & \mathsf{UpId} \mapsto \mathsf{Slot} & \text{confirmed proposals} \\ vts & \in & \mathbb{P}\ (\mathsf{UpId} \times \mathsf{VKey_G}) & \text{votes} \end{pmatrix}$$

Vote registration transitions

$$\_ \vdash \_ \xrightarrow[\text{UPVOTE}]{\_} \_ \in \mathbb{P}\ (\mathsf{VEnv} \times \mathsf{VState} \times \mathsf{Vote} \times \mathsf{VState})$$

Figure 31: Vote registration transition-system types

The rules in Figure 32 model the registration of a vote:

- The vote gets added to the list set of votes per-proposal (*vts*), via transition $\xrightarrow[\text{ADDVOTE}]{}$.
- If the number of votes for the proposal v refers to exceeds the confirmation threshold and this proposal was not confirmed already, then the proposal gets added to the set of confirmed proposals (*cps*). The reason why we check that the proposal was not already confirmed, is that we want to keep in *cps* the earliest block number in which the proposal was confirmed.

$$\frac{\begin{pmatrix} rups \\ dms \end{pmatrix} \vdash \begin{pmatrix} vts \end{pmatrix} \xrightarrow[\text{ADDVOTE}]{v} \begin{pmatrix} vts' \end{pmatrix} \qquad pid := \mathsf{vPropId}\ v \qquad (|\{pid\} \lhd vts'| < t \vee pid \in \mathrm{dom}\ cps)}{\begin{pmatrix} s_n \\ t \\ rups \\ dms \end{pmatrix} \vdash \begin{pmatrix} cps \\ vts \end{pmatrix} \xrightarrow[\text{UPVOTE}]{v} \begin{pmatrix} cps \\ vts' \end{pmatrix}} \tag{31}$$

$$\frac{\begin{pmatrix} rups \\ dms \end{pmatrix} \vdash \begin{pmatrix} vts \end{pmatrix} \xrightarrow[\text{ADDVOTE}]{v} \begin{pmatrix} vts' \end{pmatrix} \qquad pid := \mathsf{vPropId}\ v \qquad t \leq |\{pid\} \lhd vts'| \qquad pid \notin \mathrm{dom}\ cps}{\begin{pmatrix} s_n \\ t \\ rups \\ dms \end{pmatrix} \vdash \begin{pmatrix} cps \\ vts \end{pmatrix} \xrightarrow[\text{UPVOTE}]{v} \begin{pmatrix} cps \cup\!\!\!\cdot\ \{pid \mapsto s_n\} \\ vts' \end{pmatrix}} \tag{32}$$

Figure 32: Vote registration rules

6.4 Update-proposal endorsement

Figure 33 shows the types of the transition system associated with the registration of candidate protocol versions present in blocks. Some clarifications are in order:

- The k parameter is used to determine when a confirmed proposal is stable. Given we are in a current slot s_n, all update proposals confirmed at or before slot $s_n - 2 \cdot k$ are deemed stable.
- For the sake of conciseness, we omit the types associated to the transitions $\xrightarrow[\text{FADS}]{}$, since they can be inferred from the types of the $\xrightarrow[\text{UPEND}]{}$ transitions.

Update-proposal endorsement environments

$$\mathsf{BVREnv} = \left(\begin{array}{rcll} s_n & \in & \mathsf{Slot} & \text{current slot number} \\ t & \in & \mathbb{N} & \text{adoption threshold} \\ dms & \in & \mathsf{VKey_G} \mapsto \mathsf{VKey} & \text{delegation map} \\ cps & \in & \mathsf{UpId} \mapsto \mathsf{Slot} & \text{confirmed proposals} \\ rpus & \in & \mathsf{UpId} \mapsto (\mathsf{ProtVer} \times \mathsf{PParams}) & \text{registered update proposals} \end{array} \right)$$

Update-proposal endorsement states

$$\mathsf{BVRState} = \left(\begin{array}{rcll} fads & \in & (\mathsf{Slot} \times (\mathsf{ProtVer} \times \mathsf{PParams}))^* & \text{future protocol-version adoptions} \\ bvs & \in & \mathbb{P}\,(\mathsf{ProtVer} \times \mathsf{VKey_G}) & \text{endorsement-key pairs} \end{array} \right)$$

Update-proposal endorsement transitions

$$\_ \vdash \_ \xrightarrow[\text{UPEND}]{\_} \_ \in \mathbb{P}\,(\mathsf{BVREnv} \times \mathsf{BVRState} \times (\mathsf{ProtVer} \times \mathsf{VKey}) \times \mathsf{BVRState})$$

Figure 33: Update-proposal endorsement transition-system types

Rules in Figure 34 specify what happens when a block issuer signals that it is ready to upgrade to a new protocol version, given in the rule by *bv*:

- The set *bvs*, containing which genesis keys are (through their delegates) ready to adopt a given protocol version, is updated to reflect that the delegators of the block issuer (identified by its verifying key *vk*) are ready to upgrade to *bv*. Given a pair $(pv,\ vk_s) \in bvs$, we say that (the owner of) key vk_s endorses the (proposed) protocol version *pv*. Note that before the decentralized era we do not count the total number nodes that are ready to upgrade to a new protocol version, but we count only nodes that are delegated by a genesis key. This allows us to implement a simple update mechanism while we transition to the decentralized era, where we will incorporate the results of ongoing research on a decentralized update mechanism.
- If there are a significant number of genesis keys that endorse *bv* (the t environment variable is used for this), there is a registered proposal (which are contained in *rpus*) which proposes to upgrade the protocol to version *bv*, and this update proposal was confirmed at least $2 \cdot k$ slots ago (to ensure stability of the confirmation), then we update the sequence of future protocol-version adoptions (*fads*).
- An element $(s_c, (pv_c, pps_c)$ of *fads* represents the fact that protocol version pv_c got enough endorsements at slot s_c. An invariant that this sequence should maintain is that it is sorted in ascending order on slots and on protocol versions. This means that if we want to know

what is the next candidate to adopt at a slot s_k we only need to look at the last element of $[..,s_k] \lhd fads$. Since the list is sorted in ascending order on protocol versions, we know that this last element will contain the highest version to be adopted in the slot range $[..,s_k]$. The $\xrightarrow[\text{FADS}]{}$ transition rules take care of maintaining the aforementioned invariant. If a given protocol-version *bv* got enough endorsements, but there is an adoption candidate as last element of *fads* with a higher version, we simply discard *bv*.

- If a registered proposal cannot be adopted, we only register the endorsement.
- If a block version does not correspond to a registered or confirmed proposal, we just ignore the endorsement.

$$\frac{(\_;(\_,(pv_c,\_)) := fads \wedge pv_c < bv) \vee \epsilon = fads}{(\ fads\ ) \xrightarrow[\text{FADS}]{(s_n,(bv,pps_c))} (\ fads;(s_n,(bv,pps_c))\ )} \tag{33}$$

$$\frac{\_;(\_,(pv_c,\_)) := fads \qquad bv \leq pv_c}{(\ fads\ ) \xrightarrow[\text{FADS}]{(s_n,(bv,pps_c))} (\ fads\ )} \tag{34}$$

$$\frac{pid \mapsto (bv,\_) \notin rpus \vee pid \notin \text{dom}\ (cps \rhd [..,s_n - 2\cdot k])}{\begin{pmatrix} s_n \\ t \\ dms \\ cps \\ rpus \end{pmatrix} \vdash \begin{pmatrix} fads \\ bvs \end{pmatrix} \xrightarrow[\text{UPEND}]{(bv,vk)} \begin{pmatrix} fads \\ bvs \end{pmatrix}} \tag{35}$$

$$\frac{\begin{array}{c} bvs' := bvs \cup \{(bv,vk_s) \mid vk_s \mapsto vk \in dms\} \qquad |\{bv\} \lhd bvs'| < t \\ pid \mapsto (bv,\_) \in rpus \qquad pid \in \text{dom}\ (cps \rhd [..,s_n - 2\cdot k]) \end{array}}{\begin{pmatrix} s_n \\ t \\ dms \\ cps \\ rpus \end{pmatrix} \vdash \begin{pmatrix} fads \\ bvs \end{pmatrix} \xrightarrow[\text{UPEND}]{(bv,vk)} \begin{pmatrix} fads \\ bvs' \end{pmatrix}} \tag{36}$$

$$\frac{\begin{array}{c} bvs' := bvs \cup \{(bv,vk_s) \mid vk_s \mapsto vk \in dms\} \qquad t \leq |\{bv\} \lhd bvs'| \\ pid \mapsto (bv,pps_c) \in rpus \qquad pid \in \text{dom}\ (cps \rhd [..,s_n - 2\cdot k]) \\ (fads) \xrightarrow[\text{FADS}]{(s_n,(bv,pps_c))} (fads') \end{array}}{\begin{pmatrix} s_n \\ t \\ dms \\ cps \\ rpus \end{pmatrix} \vdash \begin{pmatrix} fads \\ bvs \end{pmatrix} \xrightarrow[\text{UPEND}]{(bv,vk)} \begin{pmatrix} fads' \\ bvs' \end{pmatrix}} \tag{37}$$

Figure 34: Update-proposal endorsement rules

6.5 Deviations from the `cardano-sl` implementation

The current specification of the voting mechanism deviates from the actual implementation, although it should be backwards compatible with the latter. These deviations are required to simplify the voting and update mechanism removing unnecessary features for a simplified setting, which will use the OBFT consensus protocol with federated genesis key holders. This in turn, enables us to remove any accidental complexity that might have been introduced in the current implementation. The following subsections highlight the differences between the this specification and the current implementation.

6.5.1 Positive votes

Genesis keys can only vote (positively) for an update proposal. In the current implementation stakeholders can vote for or against a proposal, which makes the voting logic more complex:

- there are more cases to consider
- the current voting validation rules allow voters to change their minds (by flipping their vote) at most once, which requires to keep track how a stake holder voted and how many times. Contrast this with Rule 30 where we only need to keep track of the set of key-proposal-id's pairs.

6.5.2 Alternative version numbers

Alternative version numbers are only lexicographically constrained. The current implementation seems to be dependent on the order in which the update proposals arrive: given a new update proposal *up*, if a set X of update proposals with the same minor and major versions than *up* exist, then the alternative version of *up* has to be one more than the maximum alternative number of X. Not only this logic seems to be brittle since it depends on the order of arrival of the update proposals, but it requires a more complex check (which depends on state) to determine if a proposed version can follow the current one. By being more lenient on the alternative versions of update proposals we can simplify the version checking logic considerably.

6.5.3 No implicit agreement

We do not model the implicit agreement rule. If a proposal does not get enough votes before the end of the voting period, then we simply discard it. At the moment it is not clear whether the implicit agreement rule is needed. Furthermore, in a non-federated setting, one could imagine an attack based on exploiting an implicit agreement rule, where the attacker would attempt to carry out a DoS attack on the parts of network that are likely to affect a proposal in a way that is undesirable for the attacker. Thus the explicit agreement seems to be a safer option.

6.5.4 Adoption threshold

The current implementation adopts a proposal with version *pv* if the portion of block issuers' stakes, which issued blocks with this version, is greater than the threshold given by:

```
max spMinThd (spInitThd - (t - s) * spThdDecrement)
```

where:

- `spMinThd` is a minimum threshold required for adoption.
- `spInitThd` is an initial threshold.
- `spThdDecrement` is the decrement constant of the initial threshold.

In this specification we only make use of a minimum adoption threshold, represented by the protocol parameter *upAdptThd* until it becomes clear why a dynamic alternative is needed.

6.5.5 No checks on unlock-stake-epoch parameter

The rule of Figure 25 does not check the `bvdUnlockStakeEpoch` parameter, since it will have a different meaning in the handover phase: its use will be reserved for unlocking the Ouroboros-BFT logic in the software.

6.5.6 Ignored attributes of proposals

In Figure 20 the types UpdData, and UpdAttrs are only needed to model the fact that an update proposal must sign such data, however, we do not use them for any other purpose in this formalization.

6.5.7 No limits on update proposals per-key per-epoch

In the current system a given genesis key can submit only one proposal per epoch. At the moment, it is not clear what are the advantages of such constraint:

- Genesis keys are controlled by the Cardano foundation.
- Even if a genesis key falls in the hands of the adversary, only one update proposal can be submitted per-block, and proposals have a time to live of u blocks. So in the worst case scenario we are looking at an increase in the state size of the ledger proportional to u.

On the other hand, having that constraint in place brings some extra complexity in the specification, and therefore in the code that will implement it. Furthermore, in the current system, if an error is made in an update proposal, then if an amendment must be made within the current epoch, then a new update proposal must be submitted with a different key, which adds extra complexity for devops. In light of the preceding discussion, unless there is a benefit for restricting the number of times a genesis key can submit an update proposal, we opted for removing such a constraint in the current specification.

6.5.8 Acceptance of blocks endorsing unconfirmed proposal updates

A consequence of enforcing the update rules in Figure 34 is that a block that is endorsing an unconfirmed proposal gets accepted, although it will not have any effect on the update mechanism. It is not clear at this stage whether such a block should be rejected, therefore we have chosen to be lenient.

6.5.9 Only genesis keys are counted for endorsement

The rules in Figure 34 take only into account the endorsements by delegates of genesis keys. The reason for this is that implementing a more complex update mechanism depends on research that is in progress at the time of writing this specification. We decided to keep the update mechanism as simple as possible in the centralized era and incorporate the research results for the decentralized era at a later stage.

7 Blockchain interface

7.1 Delegation interface

Delegation interface environments

$$\mathsf{DIEnv} = \begin{pmatrix} \mathcal{K} & \in & \mathbb{P}\,\mathsf{VKey_G} & \text{allowed delegators} \\ e & \in & \mathsf{Epoch} & \text{current epoch} \\ s & \in & \mathsf{Slot} & \text{current slot} \end{pmatrix}$$

Delegation interface states

$$\mathsf{DIState} = \begin{pmatrix} dms & \in & \mathsf{VKey_G} \mapsto \mathsf{VKey} & \text{delegation map} \\ dws & \in & \mathsf{VKey_G} \mapsto \mathsf{Slot} & \text{when last delegation occurred} \\ sds & \in & (\mathsf{Slot} \times (\mathsf{VKey_G} \times \mathsf{VKey}))^* & \text{scheduled delegations} \\ eks & \in & \mathbb{P}\,(\mathsf{Epoch} \times \mathsf{VKey_G}) & \text{key-epoch delegations} \end{pmatrix}$$

Delegation transitions

$$\_ \vdash \_ \xrightarrow[\text{DELEG}]{} \_ \in \mathbb{P}\,(\mathsf{DIEnv} \times \mathsf{DIState} \times \mathsf{DCert}^* \times \mathsf{DIState})$$

Figure 35: Delegation interface transition-system types

7.1.1 Delegation interface rules

$$\frac{\begin{pmatrix} \mathcal{K} \\ e \\ s \end{pmatrix} \vdash \xrightarrow[\text{SDELEG}]{} \begin{pmatrix} sds_0 \\ eks_0 \end{pmatrix} \qquad \mathcal{K} \vdash \xrightarrow[\text{ADELEG}]{} \begin{pmatrix} dms_0 \\ dws_0 \end{pmatrix}}{\begin{pmatrix} \mathcal{K} \\ e \\ s \end{pmatrix} \vdash \xrightarrow[\text{DELEG}]{} \begin{pmatrix} dms_0 \\ dws_0 \\ sds_0 \\ eks_0 \end{pmatrix}} \tag{38}$$

$$\frac{\begin{pmatrix} \mathcal{K} \\ e \\ s \end{pmatrix} \vdash \begin{pmatrix} sds \\ eks \end{pmatrix} \xrightarrow[\text{SDELEGS}]{\Gamma} \begin{pmatrix} sds' \\ eks' \end{pmatrix} \qquad \mathcal{K} \vdash \begin{pmatrix} dms \\ dws \end{pmatrix} \xrightarrow[\text{ADELEGS}]{[..,s] \lhd sds'} \begin{pmatrix} dms' \\ dws' \end{pmatrix}}{\begin{pmatrix} \mathcal{K} \\ e \\ s \end{pmatrix} \vdash \begin{pmatrix} dms \\ dws \\ sds \\ eks \end{pmatrix} \xrightarrow[\text{DELEG}]{\Gamma} \begin{pmatrix} dms' \\ dws' \\ [s+1,..] \lhd sds' \\ [e,..] \lhd eks' \end{pmatrix}} \tag{39}$$

Figure 36: Delegation interface rules

7.2 Update-proposals interface

Figure 38 defines the types of the transition systems related with the update-proposals interface. The acronyms in the transition labels have the following meaning:

UPIREG Update-proposal-interface registration.

UPIVOTE Update-proposal-interface vote.

UPIEND Update-proposal-interface endorsement.

UPIEC Update-proposal-interface epoch-change.

In these rules we make use of the abstract constant *ngk*, defined in Figure 37, which determines the number of genesis keys:

Abstract functions

$$\begin{array}{rcll} ngk & \in & \mathbb{N} & \text{number of genesis keys} \\ \mathsf{firstSlot} & \in & \in\ \mathsf{Epoch} \to \mathsf{Slot} & \text{first slot of an epoch} \end{array}$$

Figure 37: Update interface types and functions

Update-proposals interface environments

$$\mathsf{UPIEnv} = \left(\begin{array}{rcll} s_n & \in & \mathsf{Slot} & \text{current slot number} \\ dms & \in & \mathsf{VKey_G} \mapsto \mathsf{VKey} & \text{delegation map} \end{array} \right)$$

Update-proposals interface states

$$\mathsf{UPIState} = \left(\begin{array}{rcll} (pv, pps) & \in & \mathsf{ProtVer} \times \mathsf{PParams} & \text{current protocol information} \\ fads & \in & (\mathsf{Slot} \times (\mathsf{ProtVer} \times \mathsf{PParams}))^* & \text{future protocol version adoptions} \\ avs & \in & \mathsf{ApName} \mapsto (\mathsf{ApVer} \times \mathsf{Slot} \times \mathsf{Mdt}) & \text{application versions} \\ rpus & \in & \mathsf{UpId} \mapsto (\mathsf{ProtVer} \times \mathsf{PParams}) & \text{registered protocol update proposals} \\ raus & \in & \mathsf{UpId} \mapsto (\mathsf{ApName} \times \mathsf{ApVer} \times \mathsf{Mdt}) & \text{registered software update proposals} \\ cps & \in & \mathsf{UpId} \mapsto \mathsf{Slot} & \text{confirmed proposals} \\ vts & \in & \mathbb{P}\,(\mathsf{UpId} \times \mathsf{VKey_G}) & \text{proposals votes} \\ bvs & \in & \mathbb{P}\,(\mathsf{ProtVer} \times \mathsf{VKey_G}) & \text{endorsement-key pairs} \\ pws & \in & \mathsf{UpId} \mapsto \mathsf{Slot} & \text{proposal timestamps} \end{array} \right)$$

Update-proposals interface transitions

$$\begin{array}{rcl} \_ \vdash \_ \xrightarrow[\text{UPIREG}]{} \_ & \in & \mathbb{P}\,(\mathsf{UPIEnv} \times \mathsf{UPIState} \times \mathsf{UProp} \times \mathsf{UPIState}) \\ \_ \vdash \_ \xrightarrow[\text{UPIVOTE}]{} \_ & \in & \mathbb{P}\,(\mathsf{UPIEnv} \times \mathsf{UPIState} \times \mathsf{Vote} \times \mathsf{UPIState}) \\ \_ \vdash \_ \xrightarrow[\text{UPIEND}]{} \_ & \in & \mathbb{P}\,(\mathsf{UPIEnv} \times \mathsf{UPIState} \times (\mathsf{ProtVer} \times \mathsf{VKey}) \times \mathsf{UPIState}) \\ \_ \vdash \_ \xrightarrow[\text{UPIEC}]{} \_ & \in & \mathbb{P}\,(\mathsf{Epoch} \times \mathsf{UPIState} \times \mathsf{UPIState}) \end{array}$$

Figure 38: Update-proposals interface transition-system types

$$\frac{\begin{pmatrix} pv \\ pps \\ avs \\ dms \end{pmatrix} \vdash \begin{pmatrix} rpus \\ raus \end{pmatrix} \xrightarrow[\text{UPREG}]{up} \begin{pmatrix} rpus' \\ raus' \end{pmatrix} \qquad pws' := pws \underset{\rightarrow}{\cup} \{\mathsf{upId}\ up \mapsto s_n\}}{\begin{pmatrix} s_n \\ dms \end{pmatrix} \vdash \begin{pmatrix} (pv, pps) \\ fads \\ avs \\ rpus \\ raus \\ cps \\ vts \\ bvs \\ pws \end{pmatrix} \xrightarrow[\text{UPIREG}]{up} \begin{pmatrix} (pv, pps) \\ fads \\ avs \\ rpus' \\ raus' \\ cps \\ vts \\ bvs \\ pws' \end{pmatrix}} \tag{40}$$

Figure 39: Update-proposals registration rules

Rule 41 models the effect of voting on an update proposal.

$$\frac{upAdptThd \mapsto q \in pps \qquad \begin{pmatrix} s_n \\ \lfloor q \cdot ngk \rfloor \\ \text{dom } pws \\ dms \end{pmatrix} \vdash \begin{pmatrix} cps \\ vts \end{pmatrix} \xrightarrow[\text{UPVOTE}]{v} \begin{pmatrix} cps' \\ vts' \end{pmatrix}}{\begin{pmatrix} s_n \\ dms \end{pmatrix} \vdash \begin{pmatrix} (pv, pps) \\ fads \\ avs \\ rpus \\ raus \\ cps \\ vts \\ bvs \\ pws \end{pmatrix} \xrightarrow[\text{UPIVOTE}]{v} \begin{pmatrix} (pv, pps) \\ fads \\ avs \\ rpus \\ raus \\ cps' \\ vts' \\ bvs \\ pws \end{pmatrix}} \tag{41}$$

Figure 40: Voting on update-proposals rules

Figure 41 shows the different states in which a software proposal update might be: if valid, a software update proposal becomes active whenever it is included in a block. If the update proposal gets enough votes, then the corresponding software update proposal becomes confirmed. After this confirmation becomes stable, the new software version gets adopted. If the voting period ends without an update proposal being confirmed, then the corresponding software update proposal gets rejected. Protocol updates on the other hand, involve a slightly different logic, and the state transition diagram for these kind of updates is shown in Figure 47.

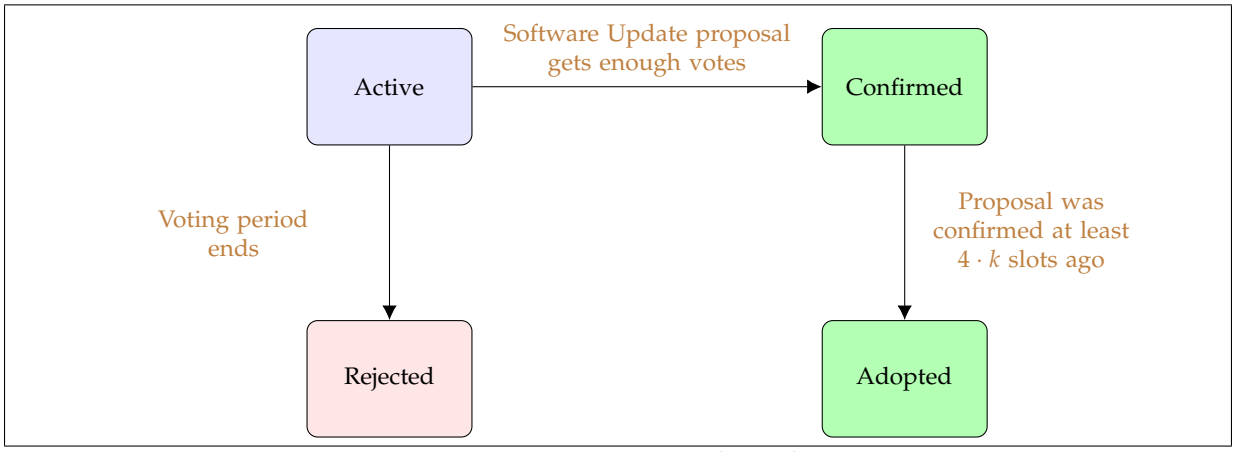

Figure 41: State-transition diagram for software-updates

A sequence of votes can be applied using $\xrightarrow[\text{UPIVOTES}]{}$ transitions. The inference rules for them are presented in Figure 42. After applying a sequence of votes, proposals might get confirmed, which means that they will be added to the set cps'. In such case, the mapping of application names to their latest version known to the ledger will be updated to include the information about the confirmed proposals. Note that, unlike protocol updates, software updates take effect as soon as a proposal is confirmed (we cannot wait for stability since we need to preserve compatibility with the existing chain, where there are software update proposals that were adopted without waiting for $2 \cdot k$ slots). In this rule, we also delete the confirmed id's from the set of registered application update proposals (*raus*), since this information is no longer needed once the application-name to software-version map (*avs*) is updated.

Also note that, unlike the rules of Figure 45, we need not remove other update proposals that refer to the software names whose versions were changed in avs_{new}. The reason for this is that the range of *raus* can contain only one pair of the form $(an, \_, \_)$ for any given application name *an* (see Rule 24).

$$\frac{}{\begin{pmatrix} s_n \\ dms \end{pmatrix} \vdash us \xrightarrow[\text{APPLYVOTES}]{\epsilon} us} \tag{42}$$

$$\frac{\begin{pmatrix} s_n \\ dms \end{pmatrix} \vdash us \xrightarrow[\text{APPLYVOTES}]{\Gamma} us' \qquad \begin{pmatrix} s_n \\ dms \end{pmatrix} \vdash us' \xrightarrow[\text{UPIVOTE}]{v} us''}{\begin{pmatrix} s_n \\ dms \end{pmatrix} \vdash us \xrightarrow[\text{APPLYVOTES}]{\Gamma;v} us''} \tag{43}$$

$$\frac{\begin{pmatrix} s_n \\ dms \end{pmatrix} \vdash \begin{pmatrix} (pv, pps) \\ fads \\ avs \\ rpus \\ raus \\ cps \\ vts \\ bvs \\ pws \end{pmatrix} \xrightarrow[\text{APPLYVOTES}]{\Gamma} \begin{pmatrix} (pv', pps') \\ fads' \\ avs' \\ rpus' \\ raus' \\ cps' \\ vts' \\ bvs' \\ pws' \end{pmatrix} \qquad \begin{array}{rcl} cfm_{raus} & := & \text{dom } (cps') \lhd raus' \\ avs_{new} & := & \{an \mapsto (av, s_n, m) \mid (an, av, m) \in cfm_{raus}\} \end{array}}{\begin{pmatrix} s_n \\ dms \end{pmatrix} \vdash \begin{pmatrix} (pv, pps) \\ fads \\ avs \\ rpus \\ raus \\ cps \\ vts \\ bvs \\ pws \end{pmatrix} \xrightarrow[\text{UPIVOTES}]{\Gamma} \begin{pmatrix} (pv', pps') \\ fads' \\ avs' \underset{\rightarrow}{\cup} avs_{new} \\ rpus' \\ \text{dom } (cps') \not\lhd raus' \\ cps' \\ vts' \\ bvs' \\ pws' \end{pmatrix}} \tag{44}$$

Figure 42: Applying multiple votes on update-proposals rules

The interface rule for protocol-version endorsement makes use of the $\xrightarrow[\text{UPEND}]{}$ transition, where we set the threshold for proposal adoption to: the number of genesis keys (ngk) times the minimum proportion of genesis keys that need to endorse an update proposal for it to become a candidate for adoption (given by the protocol parameter $upAdptThd$). In addition, the unconfirmed proposals that are older than u blocks are removed from the parts of the state that hold:

- the registered protocol and software update proposals,
- the votes associated with the proposals,
- the set of endorsement-key pairs, and
- the block number in which proposals where added.

In Rule 45, the set of proposal id's pid_{keep} contains only those proposals that haven't expired yet or that are confirmed. Once a proposal up is confirmed, it is removed from the set of confirmed proposals (cps) when a new a protocol version gets adopted (see Rule 49). The set of endorsement-key pairs is cleaned here as well as in the epoch change rule (Rule 49). The reason for this is that this set grows at each block, and it can get considerably large if no proposal gets adopted at the end of an epoch.

$$\frac{\begin{array}{c} upAdptThd \mapsto q \in pps \\ \begin{pmatrix} s_n \\ \lfloor q \cdot ngk \rfloor \\ dms \\ cps \\ rpus \end{pmatrix} \vdash \begin{pmatrix} fads \\ bvs \end{pmatrix} \xrightarrow[\text{UPEND}]{(bv,vk)} \begin{pmatrix} fads' \\ bvs' \end{pmatrix} \\ upropTTL \mapsto u \in pps \\ \begin{array}{rcl} pids_{keep} & := & \text{dom } (pws \rhd [s_n - u, ..]) \cup \text{dom } cps \\ vs_{keep} & := & \text{dom } (\text{range } rpus') \\ rpus' & := & pids_{keep} \lhd rpus \end{array} \end{array}}{\begin{pmatrix} s_n \\ dms \end{pmatrix} \vdash \begin{pmatrix} (pv, pps) \\ fads \\ avs \\ rpus \\ raus \\ cps \\ vts \\ bvs \\ pws \end{pmatrix} \xrightarrow[\text{UPIEND}]{(bv,vk)} \begin{pmatrix} (pv, pps) \\ fads' \\ avs \\ rpus' \\ pids_{keep} \lhd raus \\ cps \\ pids_{keep} \lhd vts \\ vs_{keep} \lhd bvs' \\ pids_{keep} \lhd pws \end{pmatrix}} \tag{45}$$

Figure 43: Proposal endorsement rules

Rule 49 models how the protocol-version and its parameters are changed depending on an epoch change signal. On an epoch change, this rule will pick a candidate that gathered enough endorsements at least $4 \cdot k$ slots ago. If a protocol-version candidate cannot gather enough endorsements $4 \cdot k$ slots before the end of an epoch, the proposal can only be adopted in the next epoch. The reason for the $4 \cdot k$ slot delay is to allow a period between knowing when a proposal will be adopted, and the event of its being adopted. Since update proposals can and will make large changes to the way the chain operates, it is useful to be able to guarantee a window in which it is known that no update will take place. Figure 46 shows an example of a proposal being confirmed too late in an epoch, where it is not possible to get enough endorsements in the remaining window. In this Figure we take $k = 2$, and we assume 4 endorsements are needed to consider a proposal as candidate for adoption. Note that, in the final state, we use union override to define the updated parameters ($pps \underrightarrow{\cup} pps'$). This is because candidate proposal might only update some parameters of the protocol.

In Rule 49, when a new proposal gets adopted, all the state components that refer to protocol update proposals get emptied. The reason for this is that at the moment of registering a proposal, we evaluated it in a state where the protocol parameters that we used for this are no longer up to date (see for instance Equation (22)). For instance, assume we register a proposal *up* which only changes the maximum transaction size to x, and the current block size is set to $x + 1$. Then, canUpdate holds, since the maximum transaction size is less than the maximum block size. If now a new proposal gets adopted that changes the maximum block size to $x - 1$, then this invalidates *up* since canUpdate no longer holds.

If there are no candidates for adoption, then the state variables remain unaltered (Rule 48).

Also note that the registered software-update proposals need not be cleaned here, since this is done either when a proposal gets confirmed or when it expires.

$$\frac{[.., s_n - 4 \cdot k] \lhd fads = \epsilon}{\begin{pmatrix} s_n \\ fads \end{pmatrix} \vdash \left(\ pv, pps \ \right) \xrightarrow[\text{PVBUMP}]{} \left(\ pv, pps \ \right)} \tag{46}$$

$$\frac{\_; (\_, (pv_c, pps_c)) := [.., s_n - 4 \cdot k] \lhd fads}{\begin{pmatrix} s_n \\ fads \end{pmatrix} \vdash \left(\ pv, pps \ \right) \xrightarrow[\text{PVBUMP}]{} \left(\ pv_c, pps_c \ \right)} \tag{47}$$

Figure 44: Protocol version bump rules

Figure 47 shows the different states a protocol-update proposal can be in, and what causes the transitions between them.

$$\frac{\left(\begin{array}{l}\mathsf{firstSlot}\ e_n\\ fads\end{array}\right) \vdash \left(\ pv, pps\ \right) \xrightarrow[\text{PVBUMP}]{} \left(\ pv', pps'\ \right) \qquad pv = pv'}{(e_n) \vdash \left(\begin{array}{l}(pv, pps)\\ fads\\ avs\\ rpus\\ raus\\ cps\\ vts\\ bvs\\ pws\end{array}\right) \xrightarrow[\text{UPIEC}]{} \left(\begin{array}{l}(pv, pps)\\ fads\\ avs\\ rpus\\ raus\\ cps\\ vts\\ bvs\\ pws\end{array}\right)} \tag{48}$$

$$\frac{\left(\begin{array}{l}\mathsf{firstSlot}\ e_n\\ fads\end{array}\right) \vdash \left(\ pv, pps\ \right) \xrightarrow[\text{PVBUMP}]{} \left(\ pv', pps'\ \right) \qquad pv \neq pv'}{(e_n) \vdash \left(\begin{array}{l}(pv, pps)\\ fads\\ avs\\ rpus\\ raus\\ cps\\ vts\\ bvs\\ pws\end{array}\right) \xrightarrow[\text{UPIEC}]{} \left(\begin{array}{l}(pv', pps')\\ \epsilon\\ avs\\ \emptyset\\ \emptyset\\ \emptyset\\ \emptyset\\ \emptyset\\ \emptyset\end{array}\right)} \tag{49}$$

Figure 45: Block version adoption on epoch change rules

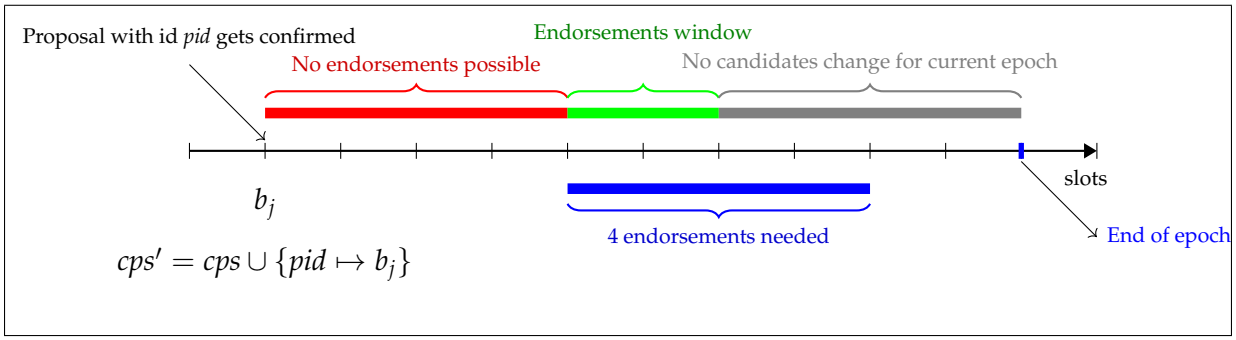

Figure 46: An update proposal confirmed too late

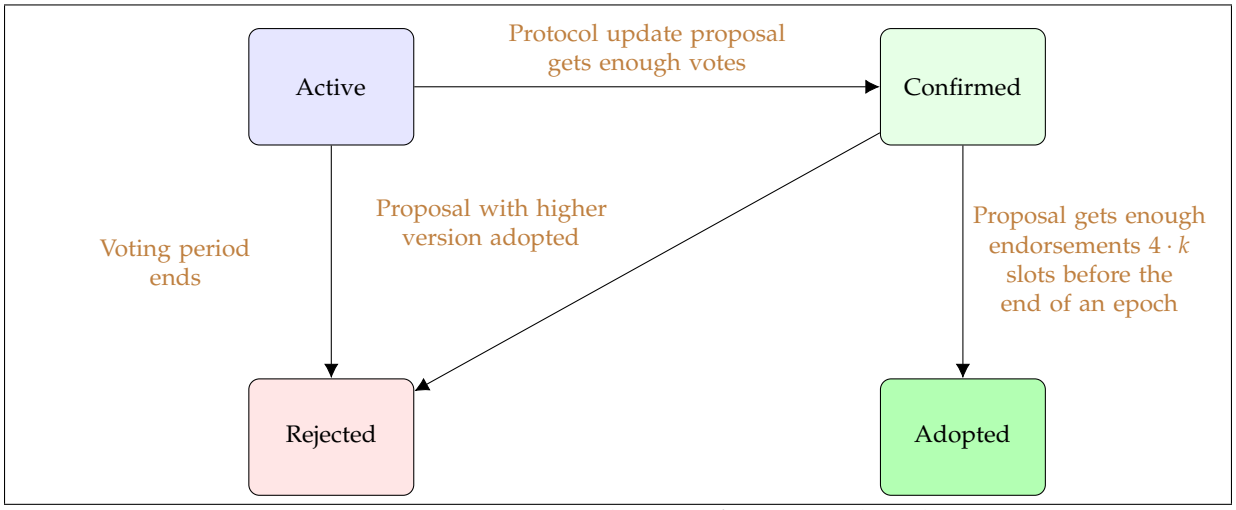

Figure 47: State-transition diagram for protocol-updates

8 Transition Systems Properties

8.1 Transition-system traces

This section introduces the notion of traces induced by the transition systems described in Formal Methods Team (2018).

Definition 2 (Traces) *Given a state transition system $L = (S, T, \Sigma, R, \Gamma)$, the set of traces of L, denoted as traces_L is defined as:*

$$\{(e, s, \mathcal{T}, s') \mid e \in \Gamma,\ s \in S,\ \mathcal{T} \in \Sigma^*,\ s' \in S\}$$

Definition 3 (Valid traces) *Given a state transition system $L = (S, T, \Sigma, R, \Gamma)$, we define the notion of valid traces inductively:*

- *For all $e \in \Gamma$, $s \in S$, (e, s, ϵ, s) is a valid trace of L.*
- *If $(e, s, \mathcal{T}, s')$ is a valid trace, and $e \vdash s' \xrightarrow{t} s''$ is a valid transition according to the rules of L, then $(e, s, \mathcal{T}; t, s'')$ is also a valid trace.*

We denote the set of valid traces of L as $\xrightarrow[\mathrm{L}]{}{}^*$, and we write $e \vdash s \xrightarrow[\mathrm{L}]{\mathcal{T}}{}^* s'$ as a shorthand for $(e, s, \mathcal{T}, s') \in \xrightarrow[\mathrm{L}]{}{}^*$. Furthermore, when the transition system name is clear from the context we will omit it from the transition arrow label.

8.2 Additional notation

We describe next additional notation needed in the properties of the transition systems specified in this document.

8.2.1 Sequence indexing

Given a sequence $\mathcal{A} \in \mathsf{A}^*$, and a natural number i such that $0 \leq i < |\mathcal{A}|$, $\mathcal{A}_i$ refers to the i^{th} element of $\mathcal{A}$.

Given a sequence $\mathcal{A}$, a quantification symbol $\bigoplus$, e.g. $\forall$ or $\exists$, a range predicate R, and a quantification term T (both of which depend on an element of $\mathcal{A}$) we write:

$$\bigoplus \mathcal{A}_i \cdot R\ \mathcal{A}_i \cdot T\ \mathcal{A}_i$$

as a shorthand notation for:

$$\bigoplus i \cdot 0 \leq i < |\mathcal{A}| \wedge R\ \mathcal{A}_i \cdot T\ \mathcal{A}_i$$

For instance:

$$\forall \mathcal{T}_i \cdot \mathsf{txins}\ \mathcal{T}_i \neq \emptyset$$

is a shorthand notation for:

$$\forall i \cdot 0 \leq i < |\mathcal{T}| \Rightarrow \mathsf{txins}\ \mathcal{T}_i \neq \emptyset$$

Remember that the range of a universal quantification can be expressed by an implication, and the range of an existential qualification by a conjunction.

8.2.2 Quantifying over set operations

Given a sequence $\mathcal{A} \in A^*$, a set B, a set operation $\bigoplus$, e.g. $\cup$ or $\underset{\rightarrow}{\cup}$, and a function $\mathsf{f} \in \mathsf{A} \to \mathsf{B}$, the term:

$$\bigoplus_{\mathsf{f}} \mathcal{A}$$

is a shorthand notation for:

$$\bigoplus_{0 \leq i < |\mathcal{A}|} \mathcal{A}_i$$

For instance:

$$\bigcup_{\mathsf{txins}} \mathcal{T}$$

denotes the sequence of unions:

$$\bigcup_{0 \leq i < |\mathcal{T}|} \mathsf{txins}\ \mathcal{T}_i$$

In a set operation quantification over a sequence $\mathcal{A}$, the operation is applied to the elements the order in which they appear in $\mathcal{A}$. This is crucial in the case of non-commutative operations, such as union override ($\underset{\rightarrow}{\cup}$).

8.3 UTxO Properties

Property 1 expresses the fact that transaction inputs cannot be used more than once. This property requires that the starting UTxO does not contain any future outputs, which is a reasonable constraint.

Property 1 (No double spending) *For all*

$$\begin{pmatrix} utxo_0 \\ reserves_0 \end{pmatrix} \xrightarrow[\text{UTXO}]{\mathcal{T}}{}^{*} \begin{pmatrix} utxo \\ reserves \end{pmatrix}$$

such that

$$\forall \mathcal{T}_i \cdot \mathrm{dom}\ (\mathsf{txouts}\ \mathcal{T}_i) \cap \mathrm{dom}\ (utxo_0) = \emptyset$$

we have:

$$\forall \mathcal{T}_i,\ \mathcal{T}_j \cdot i < j \Rightarrow \mathsf{txins}\ \mathcal{T}_i \cap \mathsf{txins}\ \mathcal{T}_j = \emptyset$$

Property 2 expresses the fact that all inputs and outputs are accounted for, in such a way that we can reconstruct the final (UTxO) state by adding all the outputs to the initial state, and removing the spent outputs.

Property 2 (UTxO is outputs minus inputs) *For all*

$$\begin{pmatrix} utxo_0 \\ reserves_0 \end{pmatrix} \xrightarrow[\text{UTXO}]{\mathcal{T}}{}^{*} \begin{pmatrix} utxo \\ reserves \end{pmatrix}$$

such that

$$\forall \mathcal{T}_i \cdot \mathrm{dom}\ (\mathsf{txouts}\ \mathcal{T}_i) \cap \mathrm{dom}\ (utxo_0) = \emptyset$$

we have:

$$\bigcup_{\mathsf{txins}} \mathcal{T} \not\triangleleft (utxo_0 \cup \bigcup_{\mathsf{txouts}} \mathcal{T}) = utxo$$

Property 3 models the fact that the amount of money in the system (counted as Lovelaces) remains constant.

Property 3 (Money supply is constant in the system) *For all*

$$\begin{pmatrix} utxo_0 \\ reserves_0 \end{pmatrix} \xrightarrow[\text{UTXO}]{\mathcal{T}}{}^{*} \begin{pmatrix} utxo \\ reserves \end{pmatrix}$$

we have:

$$reserves + \mathsf{balance}\ utxo = reserves_0 + \mathsf{balance}\ utxo_0$$

8.4 Delegation Properties

Property 4 states that delegation certificates cannot be replayed. Remember that Δ_i is the i^{th} element of Δ, which is a sequence of sequences of delegation certificates, so $\Delta_i \in \mathsf{DCert}^*$ and $\Delta_{i_j} \in \mathsf{DCert}$ (assuming j is a valid index of Δ_i).

Property 4 (No replay of delegation certificates) *For all*

$$\left(\ delegSt \ \right) \xrightarrow[\text{DELEG}]{\Delta}{}^{*} \left(\ delegSt' \ \right)$$

we have:

$$\forall i, j, k, l \cdot j < l \Rightarrow \Delta_{i_j} \neq \Delta_{k_l}$$

References

IOHK Formal Methods Team. Small step semantics for cardano, 2018. URL https://hydra.iohk.io/job/Cardano/cardano-chain/semanticsSpec/latest/download/1/small-steps.pdf.