13 Software Updates

Updates to the software will include increasing the protocol version. An increase in the major version indicates a hard fork, and the minor version a soft fork (meaning old software can validate but not produce new blocks).

The current protocol version ($\mathsf{ProtVer}$) is a member of the protocol parameters. It represents a specific version of the *ledger rules*. If pv changes, this document may have to be updated with the new rules and types if there is a change in the logic. If there is a change in the transition rules, nodes must have software installed that can implement these rules at the epoch boundary when the protocol parameter adoption occurs. Switching to using these new rules is mandatory in the sense that if the nodes do not have the applications implementing them, this will prevent a user from reading and writing to the ledger.

Applications must sometimes support *several different versions* of ledger rules in order to accommodate the timely switch of the $\mathsf{ProtVer}$ at the epoch boundary. In this situation, the newest protocol version a node is ready to use is included in the block header of the blocks it produces, see 12.1. This is either:

- the current version (if there is no protocol version update pending or the node has not updated to an upcoming software version capable of of implementing a newer protocol version), or
- the next protocol version, (if the node has updated its software, but the current protocol version on the ledger has not been updated yet).

Stake pools have some agency in the process of adoption of new protocol versions. They may refuse to download and install updates. Since software updates cannot be *forced* on the users, if the majority of users do not perform an update which allows the switch to the next $\mathsf{ProtVer}$, it cannot happen.

Note that if there is a *new protocol version* implemented by new software, the core nodes can monitor how many nodes are ready to use the new protocol version via the block headers. Once enough nodes are ready for the new protocol version, this may now be updated as well (by the mechanism in described in Section 7).

14 Transition Rule Dependencies

Figure 78 shows all STS rules, the sub-rules they use and possible dependencies. Each node in the graph represents one rule, the top rule being CHAIN. A straight arrow from one node to another one represents a sub-rule relationship. There are two recursive rules, LEDGERS and DELEGS which have self loops.

An arrow with a dotted line from one node to another represents a dependency in the sense that the output of the target rule is an input to the source one, either as part of the source state, the environment or the signal. In most cases these dependencies are between sub-rules of a rule. In the case of recursive rules, the sub-rule can also have a dependency on the super-rule. Those recursively call themselves while traversing the input signal sequence until reaching the base case with an empty input sequence.

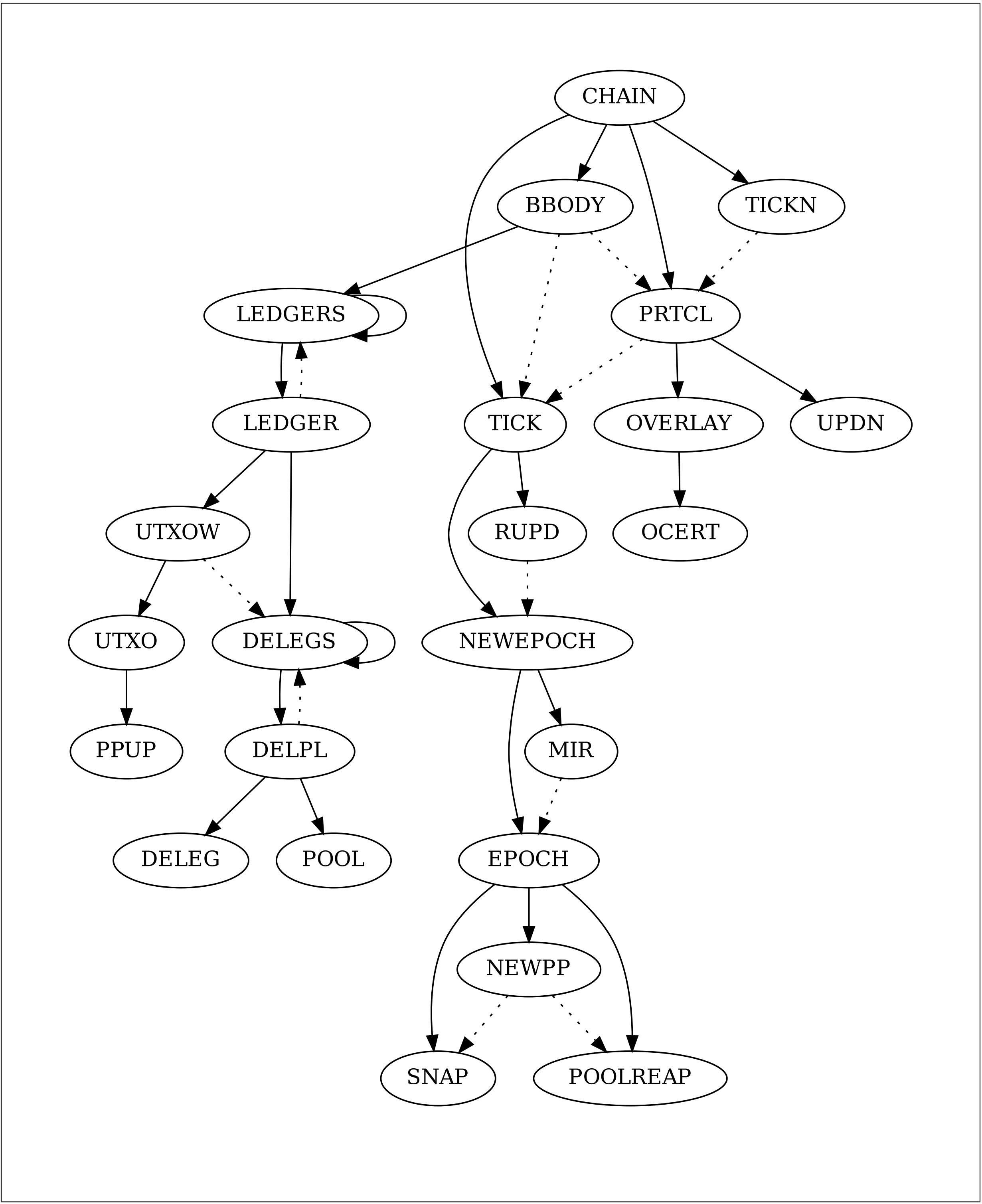

Figure 78: STS Rules, Sub-Rules and Dependencies

15 Properties

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

15.1 Header-Only Validation

The header-only validation properties of the Shelley Ledger are the analogs of those from Section 8.1 of [BC-D1].

In any given chain state, the consensus layer needs to be able to validate the block headers without having to download the block bodies. Property 15.1 states that if an extension of a chain that spans less than StabilityWindow slots is valid, then validating the headers of that extension is also valid. This property is useful for its converse: if the header validation check for a sequence of headers does not pass, then we know that the block validation that corresponds to those headers will not pass either.

First we define the header-only version of the CHAIN transition, which we call CHAINHEAD. It is very similar to CHAIN, the differences being:

- The CHAINHEAD signal is not a block, but a block header (BHeader).
- CHAINHEAD does not call BBODY.
- CHAINHEAD does not call TICK, but instead calls the similar TICKF, which differs by not calling the reward update transition RUPD.
- CHAINHEAD does not store the new epoch state *nes* in its state, but rather contains it in the environment. We will conveniently **abuse the tuple notation** and write $(nes, \tilde{s}) = s$ for splitting the chain state into the new epoch state and the remaiing fields.

$$\text{TickForecast}\ \frac{\begin{array}{c}\vdash nes \xrightarrow[\text{NEWEPOCH}]{\mathsf{epoch}\ slot} nes' \\[1ex] (e'_\ell,\ b'_{prev},\ b'_{cur},\ es',\ ru',\ pd') := nes' \\ es'' := \mathsf{adoptGenesisDelegs}\ es'\ slot \\ forecast := (e'_\ell,\ b'_{prev},\ b'_{cur},\ es'',\ ru',\ pd')\end{array}}{\vdash nes \xrightarrow[\text{TICKF}]{slot} \mathit{forecast}} \qquad (44)$$

Figure 79: Tick Forecast rules

$$\text{ChainHead}\ \dfrac{\begin{array}{c}
bhb := \mathsf{bhbody}\ bh \qquad\qquad s := \mathsf{bslot}\ bhb \\[1ex]
\mathsf{prtlSeqChecks}\ lab\ bh \\[1ex]
\vdash nes \xrightarrow[\text{TICKF}]{s} forecast \\[1ex]
(e_1,\ \_,\ \_,\ \_,\ \_,\ \_) := nes \\
(e_2,\ \_,\ \_,\ es,\ \_,\ \_,\ pd) := forecast \\
(acnt,\ \_, ls,\ \_,\ pp) := es \\
(\_, ((\_,\ \_,\ \_,\ \_,\ genDelegs,\ \_),\ (\_,\ \_,\ \_))) := ls \\
ne := e_1 \neq e_2 \\
\eta_{ph} := \mathsf{prevHashToNonce}\ (\mathsf{lastAppliedHash}\ lab) \\[1ex]
\mathsf{chainChecks}\ \mathsf{MaxMajorPV}\ (\mathsf{maxHeaderSize}\ pp,\ \mathsf{maxBlockSize}\ pp,\ \mathsf{pv}\ pp)\ bh \\[1ex]
\begin{array}{c} pp \\ \eta_c \\ \eta_{ph} \end{array} \vdash \begin{pmatrix} \eta_0 \\ \eta_h \end{pmatrix} \xrightarrow[\text{TICKN}]{ne} \begin{pmatrix} \eta_0' \\ \eta_h' \end{pmatrix} \\[1ex]
\begin{array}{c} (\mathsf{d}\ pp) \\ pd \\ genDelegs \\ \eta_0' \end{array} \vdash \begin{pmatrix} cs \\ \eta_v \\ \eta_c \end{pmatrix} \xrightarrow[\text{PRTCL}]{bh} \begin{pmatrix} cs' \\ \eta_v' \\ \eta_c' \end{pmatrix} \\[1ex]
lab' := (\mathsf{bblockno}\ bhb,\ s,\ \mathsf{bhash}\ bh)
\end{array}}{nes \vdash \begin{pmatrix} cs \\ \eta_0 \\ \eta_v \\ \eta_c \\ \eta_h \\ lab \end{pmatrix} \xrightarrow[\text{CHAINHEAD}]{bh} \begin{pmatrix} \mathbf{cs'} \\ \boldsymbol{\eta_0'} \\ \boldsymbol{\eta_v'} \\ \boldsymbol{\eta_c'} \\ \boldsymbol{\eta_h'} \\ \mathbf{lab'} \end{pmatrix}} \quad (45)$$

Figure 80: Chain-Head rules

Property 15.1 (Header only validation)**.** For all states s with slot number t[1], and chain extensions E with corresponding headers H such that:

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

we have:

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

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

Property 15.2 (Body only validation)**.** For all states s with slot number t, and chain extensions $E = [b_0, \ldots, b_n]$ with corresponding headers H such that:

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

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