Staking Contract Properties

Andreas Lynge

September 4, 2025

Abstract

This document defines the properties of a mathematical model of the staking contract and serves as the test specification for a staking contract fuzz tester. Section 1 describes the structure of the contract in Solidity-style pseudocode. Section 2 introduces the notation and the model. Section 3 states the invariants that every valid state must satisfy, and Section 4 specifies pre-conditions and post-conditions for the contract's interface functions.

Contents

1 The Staking Contract

This section describes the structure of the staking contract using a Solidity-style pseudocode notation.

1.1 Types

The staking contract makes use of the following struct types.

```
struct ListNode {
    uint64  inext;
    uint64  iprev;
    address anext;
    address aprev;
}

struct Delegator {
    uint256  accumulated_reward_per_token;
    uint256  rewards;
    uint256  stake;
    uint256  delta_stake;
    uint256  next_delta_stake;
    uint64   delta_epoch;
    uint64   next_delta_epoch;
    ListNode list_node;
}

struct KeysPacked {
    bytes secp_pubkey;
    bytes bls_pubkey;
}

struct ValExecution {
```

```
    KeysPacked keys;
    address    auth_address;
    uint64     flags;
    uint256    accumulated_reward_per_token;
    uint256    stake;
    uint256    commission;
    uint256    unclaimed_rewards;
}

struct ConsensusView {
    uint256 stake;
    uint256 commission;
}

struct WithdrawalRequest {
    uint256 amount;
    uint256 acc;
    uint64  epoch;
}
```

1.2 Fields

The staking contract has the following fields.

```
// Current epoch count (affects the active validator set and stake)
uint64 epoch;

// Whether the staking contract is in the epoch delay period
// (affects when new stake delegations are activated)
bool in_epoch_delay_period;

// Validator ID of the most recently added validator
// (incremented for each new validator)
uint64 last_val_id;

// Set of validators that are potential candidates to be active
// in the next epoch
uint64[] valset_execution;

// Set of validators that have been chosen as active, in this or
// the next epoch, depending on the value of in_epoch_delay_period
uint64[] valset_consensus;

// The previous valset_consensus. If in_epoch_delay_period is true,
// then valset_snapshot is the set of currently active validators;
// otherwise valset_consensus is the set of active validators.
uint64[] valset_snapshot;

// Auth address to corresponding validator ID
mapping(address => uint64) val_id;

// Validator ID of the proposer of each block
```

```
uint64 proposer_val_id;

// Address of BLS public key to corresponding validator ID
mapping(address => uint64) val_id_bls;

// Bit set corresponding to valset_execution
mapping(uint64 => uint256) val_bitset_bucket;

// Validator ID to validator state
mapping(uint64 => ValExecution) val_execution;

// Validator ID to state related to validators in valset_consensus
mapping(uint64 => ConsensusView) consensus_view;

// Validator ID to state related to validators in valset_snapshot
mapping(uint64 => ConsensusView) snapshot_view;

// Validator ID and address to corresponding delegator state
mapping(uint64 => mapping(address => Delegator)) delegator;

// Validator ID, address, and withdrawal ID to corresponding
// withdrawal request
mapping(uint64 =>
    mapping(address =>
        mapping(uint8 => WithdrawalRequest))) withdrawal_request;

// Epoch and validator ID to the accumulated reward per token for
// the given validator in the given epoch
mapping(uint64 => mapping(uint64 => uint256)) accumulated_reward_per_token;
```

1.3 Precompile Interface

```
function precompile_get_validator(uint64 val_id)
external view returns (
    address auth_address,
    uint64  flags,
    uint256 stake,
    uint256 accumulated_reward_per_token,
    uint256 commission,
    uint256 unclaimed_rewards,
    uint256 consensus_view_stake,
    uint256 consensus_view_commission,
    uint256 snapshot_view_stake,
    uint256 snapshot_view_commission
);

function precompile_get_delegator(uint64 val_id, address delegator)
external /* not view */ returns (
    uint256 stake,
    uint256 accumulated_reward_per_token,
    uint256 rewards,
```

```
    uint256 delta_stake,
    uint256 next_delta_stake,
    uint64  delta_epoch,
    uint64  next_delta_epoch
);

function precompile_get_withdrawal_request(uint64 val_id, address delegator)
external view returns (uint256 amount, uint256 acc, uint64 epoch);

function precompile_get_consensus_valset(uint32 start_index)
external view returns (bool done, uint32 next_index, uint64[]);

function precompile_get_snapshot_valset(uint32 start_index)
external view returns (bool done, uint32 next_index, uint64[]);

function precompile_get_execution_valset(uint32 start_index)
external view returns (bool done, uint32 next_index, uint64[]);

function precompile_get_delegations(address delegator, uint64 start_val_id)
external view returns (bool done, uint64 next_val_id, uint64[]);

function precompile_get_delegators(uint64 val_id, address start_delegator)
external view returns (bool done, address next_delegator, address[]);

function precompile_get_epoch() external view returns (uint64);

function precompile_fallback() external pure;

function precompile_add_validator(
    bytes message,
    bytes secp_signature,
    bytes bls_signature
) external payable returns (uint64);

function precompile_delegate(uint64 val_id) external payable returns (bool);

function precompile_undelegate(uint64 val_id, uint256 stake)
external returns (bool);

function precompile_compound(uint64 val_id) external returns (bool);

function precompile_withdraw(uint64 val_id, uint8 withdrawal_id)
external returns (bool);

function precompile_claim_rewards(uint64 val_id) external returns (bool);

function precompile_change_commission(uint64 val_id, uint256 new_commission)
external returns (bool);

function precompile_external_reward(uint64 val_id)
external payable returns (bool);
```

1.4 Syscall Interface

```
function syscall_on_epoch_change(uint64 next_epoch) external;

function syscall_reward(address block_author, uint256 raw_reward) external;

function syscall_snapshot() external;

function distribute_priority_fees(uint256 fees) external;
```

2 Model and Notation

2.1 Notation

The model uses the following notation for basic sets.

- The set of all 20-byte Ethereum addresses is modeled as the integers modulo 2^{160},

$$\texttt{address} = \mathbb{Z}/2^{160}\mathbb{Z} = \{0, 1, \ldots, 2^{160} - 1\}.$$

- The set of all 32-byte Ethereum words is the integers modulo 2^{256},

$$\texttt{uint256} = \mathbb{Z}/2^{256}\mathbb{Z} = \{0, 1, \ldots, 2^{256} - 1\}.$$

- The set of all 8-byte values is $\texttt{uint64} = \mathbb{Z}/2^{64}\mathbb{Z} = \{0, 1, \ldots, 2^{64} - 1\}$.
- The set of all 1-byte values is $\texttt{uint8} = \mathbb{Z}/2^{8}\mathbb{Z} = \{0, 1, \ldots, 2^{8} - 1\}$.
- The set containing two elements is modeled as $\texttt{bool} = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$.

Addition, multiplication, and additive inverse in $\mathbb{Z}/m\mathbb{Z}$ are defined as the usual ring operations. Division of $x, y \in \mathbb{Z}/m\mathbb{Z}$ with $y \neq 0$ is defined by real division rounded down, as is typical in programming languages,

$$\frac{x}{y} = \left\lfloor \frac{i(x)}{i(y)} \right\rfloor,$$

where $\lfloor \cdot \rfloor : \{x \in \mathbb{R} \mid 0 \leq x < m\} \to \mathbb{Z}/m\mathbb{Z}$ is the floor function and $i : \mathbb{Z}/m\mathbb{Z} \to \mathbb{R}$ is the canonical inclusion.

Structs are modeled as products, and dot notation is used for the projections. For example, the `ConsensusView` struct type is modeled as the product

$$\texttt{ConsensusView} = \texttt{uint256} \times \texttt{uint256}$$

with projections

$$x.\texttt{stake} \in \texttt{uint256}, \qquad x.\texttt{commission} \in \texttt{uint256},$$

for $x \in \texttt{ConsensusView}$.

An array is modeled as a length together with a finite sequence. For example, the array type `uint256[]` is modeled by the set of pairs of the form

$$\left(n, (a_i)_{0 \le i < n}\right) \in \texttt{uint256[]},$$

where $n \in \texttt{uint64}$ and $a_i \in \texttt{uint256}$. With $x = \left(n, (a_i)_{0 \le i < n}\right) \in \texttt{uint256[]}$, define

$$x.\texttt{length} = n, \qquad x_i = a_i \quad \text{for } 0 \le i < n.$$

The `bytes` Solidity type is modeled by a `uint8` array,

$$\texttt{bytes} = \texttt{uint8[]}.$$

Mappings are modeled as functions. For instance,

$$\begin{aligned}&\texttt{mapping(uint64 => mapping(address => Delegator))}\\ &\quad = \left\{ f : \texttt{uint64} \times \texttt{address} \to \texttt{Delegator} \right\},\end{aligned}$$

that is, this mapping type is modeled by the set of all binary functions from $\texttt{uint64} \times \texttt{address}$ to `Delegator`.

The set `State` denotes the set of all valid Monad blockchain states. For any $s \in \texttt{State}$, the staking contract fields under s satisfy the invariants defined in Section 3, and the staking contract interface functions satisfy the properties defined in Section 4.

The staking contract fields are generally modeled with the same names as in the Solidity pseudocode and are accessed via dot notation. For example, the field

```
mapping(address => uint64) val_id;
```

is a projection of `State`,

$$s.\texttt{val\_id} : \texttt{address} \to \texttt{uint64}, \quad \text{for } s \in \texttt{State}.$$

The `precompile_*` staking contract interface functions are modeled as functions with the same names. For example,

```
function precompile_get_delegations(address delegator, uint64 start_val_id)
external view returns (bool done, uint64 next_val_id, uint64[]);
```

is modeled as a partial function

$$\begin{aligned}\texttt{precompile\_get\_delegations} : \texttt{address} \times \texttt{uint64} \times \texttt{address} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \\ \texttt{bool} \times \texttt{uint64} \times \texttt{uint64[]} \times \texttt{State}.\end{aligned}$$

The last three arguments, $\texttt{address} \times \texttt{uint256} \times \texttt{State}$, model the sender address, the message value, and the input state of the Monad blockchain, respectively. The remaining arguments, $\texttt{address} \times \texttt{uint64}$, are translated directly from the input type of the staking contract interface function. The last output, `State`, is the resulting Monad blockchain state, and the remaining

outputs, `bool` $\times$ `uint64` $\times$ `uint64[]`, are translated directly from the output type of the staking contract interface function.

The syscall staking contract interface functions are modeled in a similar way, except that their arguments do *not* include a sender address or a message value. For example, the

```
function syscall_on_epoch_change(uint64 next_epoch) external;
```

staking contract interface function is modeled by a partial function

$$\texttt{syscall\_on\_epoch\_change} : \texttt{uint64} \times \texttt{State} \rightharpoonup \texttt{State}.$$

For any $s \in \texttt{State}$,

$$s.\texttt{balance\_of} : \texttt{address} \to \texttt{uint256}$$

is the function returning the balance of the given address in state s.

Finally, for readability, the field name `accumulated_reward_per_token` is abbreviated as `acc_rpt` in mathematical formulas; the pseudocode listings retain the full name.

2.2 State Extensions

The model of the staking contract has fields in addition to the fields of the C++ code. In Solidity pseudocode notation, these extra fields are:

```
// An upper bound on reward rounding errors
uint256 error_bound;

// unit_bias_rewards(v, a) is the sum of rewards distributed to
// delegator (v, a) but not yet claimed or compounded
mapping(uint64 => mapping(address => uint256)) unit_bias_rewards;

// The state of the active consensus validators
mapping(uint64 => ConsensusView) active_consensus_view;

// delegator_stake(v, a, e) is the stake of delegator (v, a)
// in epoch e
mapping(uint64 =>
    mapping(address =>
        mapping(uint64 => uint256))) delegator_stake;

// withdrawal_stake(v, a, e) is the stake of delegator (v, a) in
// epoch e which has been undelegated but is still eligible for
// rewards
mapping(uint64 =>
    mapping(address =>
        mapping(uint64 => uint256))) withdrawal_stake;
```

These fields of the model are explicitly updated by the post-conditions of the partial interface functions, which are defined in Section 4. The fields are modeled as projections of the state, analogously to the other fields:

- s.error_bound $\in$ uint256
- s.unit_bias_rewards : uint64 $\times$ address $\to$ uint256
- s.active_consensus_view : uint64 $\to$ ConsensusView
- s.delegator_stake : uint64 $\times$ address $\times$ uint64 $\to$ uint256
- s.withdrawal_stake : uint64 $\times$ address $\times$ uint64 $\to$ uint256

where $s \in$ State.

2.3 Constants

- STAKING_CA $\in$ address refers to the staking contract address. It is used in particular to access the balance of the staking contract.
- UNIT_BIAS $\in$ uint256 is a constant value used in the staking contract to limit rounding errors in MON rewards.
- ACTIVE_VALSET_SIZE $\in$ uint64 is an upper bound on the number of active validators.
- ACTIVE_VALIDATOR_STAKE $\in$ uint256 is a lower bound on the stake of active validators.
- MIN_VALIDATE_STAKE $\in$ uint256 is the threshold below which a validator is considered withdrawn.
- The constant flags
  - ValidatorFlagsStakeTooLow $= 1 \in$ uint64 and
  - ValidatorFlagWithdrawn $= 2 \in$ uint64

  are used to indicate validator status.
- DUST_THRESHOLD $\in$ uint256 is the minimum non-zero delegator stake.
- WITHDRAWAL_DELAY $\in$ uint64 is the number of epochs a delegator must wait before withdrawing.
- MON $\in$ uint256 is defined by MON $= 10^{18}$.
- MAX_COMMISSION $=$ MON is the upper bound on commission.
- MIN_EXTERNAL_REWARD, MAX_EXTERNAL_REWARD $\in$ uint256 are, respectively, a lower and an upper bound on external rewards.

2.4 Utilities

The relation BitTest $\subseteq (\mathbb{Z}/2^n\mathbb{Z}) \times (\mathbb{Z}/2^n\mathbb{Z})$, for $n \in \mathbb{N}$, is defined by: BitTest(x, b) holds if and only if

- $b = 2^m$ for some $m \in \mathbb{N}_0$, and
- $q(x) \geq b$, for $q : \mathbb{Z} \to \mathbb{Z}/2b\mathbb{Z}$ the quotient map.

Intuitively, the BitTest relation can be used to test whether b has exactly one bit set and

whether that bit is also set in x.

The following four functions are important for the solvency invariants and reward distributions:

- $\texttt{calculate\_rewards} : \texttt{uint256}^3 \to \texttt{uint256}$,
- $\texttt{withdrawal\_reward} : \texttt{uint64} \times \texttt{address} \times \texttt{uint8} \times \texttt{State} \to \texttt{uint256}$,
- $\texttt{unaccumulated\_rewards} : \texttt{uint64} \times \texttt{address} \times \texttt{State} \to \texttt{uint256}$, and
- $\texttt{pending\_rewards} : \texttt{uint64} \times \texttt{address} \times \texttt{State} \to \texttt{uint256}$,

defined as follows. First,

$$\texttt{calculate\_rewards}(x, q, p) = \frac{x(q-p)}{\texttt{UNIT\_BIAS}}.$$

Second,

$$\texttt{withdrawal\_reward}(v, a, i, s) = \begin{cases} \texttt{calculate\_rewards}(w.\texttt{amount}, q, w.\texttt{acc}) & \text{if } w.\texttt{epoch} > 0, \\ 0 & \text{if } w.\texttt{epoch} = 0, \end{cases}$$

where

- $w = s.\texttt{withdrawal\_request}(v, a, i)$ and
- $q = \begin{cases} s.\texttt{val\_execution}(v).\texttt{acc\_rpt} & \text{if } s.\texttt{epoch} < w.\texttt{epoch}, \\ s.\texttt{acc\_rpt}(w.\texttt{epoch}, v) & \text{if } s.\texttt{epoch} \geq w.\texttt{epoch}. \end{cases}$

Third,

$$\texttt{unaccumulated\_rewards}(v, a, s) = r_1 + r_2 + r_3,$$

where, writing $D = s.\texttt{delegator}(v, a)$ for brevity:

- $t_1 = D.\texttt{stake}$
- $p_1 = D.\texttt{acc\_rpt}$
- $d_1 = s.\texttt{acc\_rpt}(D.\texttt{delta\_epoch}, v)$
- $q_1 = \begin{cases} p_1 & \text{if } \delta = 0 \text{ or } \delta > s.\texttt{epoch} \\ d_1 & \text{if } \delta \neq 0 \text{ and } \delta \leq s.\texttt{epoch} \end{cases}$ for $\delta = D.\texttt{delta\_epoch}$
- $r_1 = \texttt{calculate\_rewards}(t_1, q_1, p_1)$
- $t_2 = t_1 + \begin{cases} D.\texttt{delta\_stake} & \text{if } \delta \leq s.\texttt{epoch} \\ 0 & \text{if } \delta > s.\texttt{epoch} \end{cases}$ for $\delta = D.\texttt{delta\_epoch}$
- $p_2 = q_1$
- $d_2 = s.\texttt{acc\_rpt}(D.\texttt{next\_delta\_epoch}, v)$
- $q_2 = \begin{cases} p_2 & \text{if } \delta = 0 \text{ or } \delta > s.\texttt{epoch} \\ d_2 & \text{if } \delta \neq 0 \text{ and } \delta \leq s.\texttt{epoch} \end{cases}$ for $\delta = D.\texttt{next\_delta\_epoch}$
- $r_2 = \texttt{calculate\_rewards}(t_2, q_2, p_2)$

- $t_3 = t_2 + \begin{cases} D.\texttt{next\_delta\_stake} & \text{if } \delta \leq s.\texttt{epoch} \\ 0 & \text{if } \delta > s.\texttt{epoch} \end{cases}$ for $\delta = D.\texttt{next\_delta\_epoch}$
- $p_3 = q_2$
- $q_3 = s.\texttt{val\_execution}(v).\texttt{acc\_rpt}$
- $r_3 = \texttt{calculate\_rewards}(t_3, q_3, p_3)$

Finally, the `pending_rewards` function is defined by

$$\texttt{pending\_rewards}(v, a, s) = \texttt{unaccumulated\_rewards}(v, a, s) + \sum_{i \in \texttt{uint8}} \texttt{withdrawal\_reward}(v, a, i, s).$$

3 Invariants

Let $s \in \texttt{State}$ be any state. The invariants defined in this section are satisfied by s. Unless quantified otherwise, the invariants hold for all $v \in \texttt{uint64}$, $a \in \texttt{address}$, and $i \in \texttt{uint8}$.

3.1 Valset Invariants

- $s.\texttt{valset\_execution}$ consists of pairwise distinct elements.
- If v is in $s.\texttt{valset\_execution}$, then $s.\texttt{val\_execution}(v).\texttt{auth\_address} \neq 0$.
- $s.\texttt{valset\_consensus}$ consists of pairwise distinct elements.
- $s.\texttt{valset\_consensus.length} \leq \texttt{ACTIVE\_VALSET\_SIZE}$.
- $s.\texttt{valset\_snapshot}$ consists of pairwise distinct elements.
- $s.\texttt{valset\_snapshot.length} \leq \texttt{ACTIVE\_VALSET\_SIZE}$.
- If v is in $s.\texttt{valset\_consensus}$, then v is in $s.\texttt{valset\_execution}$.
- v is in $s.\texttt{valset\_consensus}$ if and only if $s.\texttt{consensus\_view}(v).\texttt{stake} \geq \texttt{ACTIVE\_VALIDATOR\_STAKE}$.
- v is in $s.\texttt{valset\_snapshot}$ if and only if $s.\texttt{snapshot\_view}(v).\texttt{stake} \geq \texttt{ACTIVE\_VALIDATOR\_STAKE}$.
- $s.\texttt{consensus\_view}(v).\texttt{commission} \leq \texttt{MAX\_COMMISSION}$.
- $s.\texttt{snapshot\_view}(v).\texttt{commission} \leq \texttt{MAX\_COMMISSION}$.
- $\texttt{BitTest}(s.\texttt{val\_bitset\_bucket}(v), b)$ holds if and only if v is in $s.\texttt{valset\_execution}$, where $b = 2^r$ and r is the remainder after integer division of $i(v)$ by 256, for i the inclusion $\texttt{uint64} \to \mathbb{Z}$.
- If both $s.\texttt{val\_execution}(v).\texttt{stake} \geq \texttt{ACTIVE\_VALIDATOR\_STAKE}$ and

  $$d.\texttt{stake} + d.\texttt{delta\_stake} + d.\texttt{next\_delta\_stake} \geq \texttt{MIN\_VALIDATE\_STAKE},$$

  where $d = s.\texttt{delegator}(v, s.\texttt{val\_execution}(v).\texttt{auth\_address})$, then v is an element of $s.\texttt{valset\_execution}$.

- $s.\texttt{in\_epoch\_delay\_period} = 0$ implies $s.\texttt{active\_consensus\_view}(v) = s.\texttt{consensus\_view}(v)$ for all $v \in \texttt{uint64}$.
- $s.\texttt{in\_epoch\_delay\_period} = 1$ implies $s.\texttt{active\_consensus\_view}(v) = s.\texttt{snapshot\_view}(v)$ for all $v \in \texttt{uint64}$.

3.2 ValExecution Invariants

- For $1 \leq v \leq s.\texttt{last\_val\_id}$, $s.\texttt{val\_execution}(v).\texttt{auth\_address} \neq 0$.
- For $v > s.\texttt{last\_val\_id}$ or $v = 0$, $s.\texttt{val\_execution}(v).\texttt{auth\_address} = 0$.
- For $v \geq 1$, if $s.\texttt{val\_execution}(v+1).\texttt{auth\_address} \neq 0$, then $s.\texttt{val\_execution}(v).\texttt{auth\_address} \neq 0$.
- For all $0 < v \leq s.\texttt{last\_val\_id}$ with $a = s.\texttt{val\_execution}(v).\texttt{auth\_address}$,

  $$s.\texttt{delegator}(v,a).\texttt{stake} + s.\texttt{delegator}(v,a).\texttt{delta\_stake} + s.\texttt{delegator}(v,a).\texttt{next\_delta\_stake} \geq \texttt{MIN\_VALIDATE\_STAKE}$$

  if and only if $\texttt{BitTest}(s.\texttt{val\_execution}(v).\texttt{flags}, \texttt{ValidatorFlagWithdrawn})$ is false.
- For all $v \in \texttt{uint64}$,
  - $s.\texttt{val\_execution}(v).\texttt{commission} \leq \texttt{MAX\_COMMISSION}$ and
  - $s.\texttt{val\_execution}(v).\texttt{flags} < 4$.
- For $0 < v \leq s.\texttt{last\_val\_id}$, $s.\texttt{val\_execution}(v).\texttt{stake} \geq \texttt{ACTIVE\_VALIDATOR\_STAKE}$ if and only if $\texttt{BitTest}(s.\texttt{val\_execution}(v).\texttt{flags}, \texttt{ValidatorFlagsStakeTooLow})$ is false.
- For $v \in \texttt{uint64}$, $s.\texttt{val\_id}(y) \neq 0$ if and only if $1 \leq v \leq s.\texttt{last\_val\_id}$, for y the SECP address of the SECP public key $s.\texttt{val\_execution}(v).\texttt{keys.secp\_pubkey}$.
- For $v \in \texttt{uint64}$, $s.\texttt{val\_id\_bls}(z) \neq 0$ if and only if $1 \leq v \leq s.\texttt{last\_val\_id}$, for z the BLS address of the BLS public key $s.\texttt{val\_execution}(v).\texttt{keys.bls\_pubkey}$.

3.3 Delegator Invariants

For all $v \in \texttt{uint64}$, $a \in \texttt{address}$, and $i \in \texttt{uint8}$:

- $s.\texttt{delegator}(v,a).\texttt{next\_delta\_stake} = 0$ or $s.\texttt{delegator}(v,a).\texttt{next\_delta\_stake} \geq \texttt{DUST\_THRESHOLD}$.
- $s.\texttt{delegator}(v,a).\texttt{delta\_stake} = 0$ or $s.\texttt{delegator}(v,a).\texttt{delta\_stake} \geq \texttt{DUST\_THRESHOLD}$.
- $s.\texttt{delegator}(v,a).\texttt{stake} = 0$ or $s.\texttt{delegator}(v,a).\texttt{stake} \geq \texttt{DUST\_THRESHOLD}$.
- $s.\texttt{delegator}(v,a).\texttt{next\_delta\_stake} = 0$ if and only if $s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch} = 0$.
- $s.\texttt{delegator}(v,a).\texttt{delta\_stake} = 0$ if and only if $s.\texttt{delegator}(v,a).\texttt{delta\_epoch} = 0$.
- If $s.\texttt{delegator}(v,a).\texttt{delta\_epoch} \neq 0$ and $s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch} \neq 0$,

then

$$s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch} = 1 + s.\texttt{delegator}(v,a).\texttt{delta\_epoch}.$$

- $s.\texttt{delegator}(v,a).\texttt{delta\_epoch} \leq s.\texttt{epoch} + 1$.
- $s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch} \leq s.\texttt{epoch} + 2$.
- $s.\texttt{withdrawal\_request}(v,a,i).\texttt{amount} = 0$ if and only if $s.\texttt{withdrawal\_request}(v,a,i).\texttt{epoch} = 0$, for all v, a, i.
- With $r = s.\texttt{delegator}(v,a).\texttt{rewards} + \texttt{pending\_rewards}(v,a,s)$ and $E = s.\texttt{error\_bound} + 3 + 256$,

$$\begin{aligned} s.\texttt{unit\_bias\_rewards}(v,a) &\leq (r+E) \cdot \texttt{UNIT\_BIAS} \\ &\leq s.\texttt{unit\_bias\_rewards}(v,a) + E \cdot \texttt{UNIT\_BIAS}. \end{aligned}$$

  Note that the additional $3 + 256$ term accounts for potential rounding errors, in wei, from `pending_rewards`.
- $s.\texttt{delegator\_stake}(v,a,s.\texttt{epoch}) = s.\texttt{delegator}(v,a).\texttt{stake} + d_1 + d_2$, where
  - $\delta_1 = s.\texttt{delegator}(v,a).\texttt{delta\_epoch}$,
  - $d_1 = \begin{cases} s.\texttt{delegator}(v,a).\texttt{delta\_stake} & \text{if } \delta_1 \leq s.\texttt{epoch} \\ 0 & \text{if } \delta_1 > s.\texttt{epoch} \end{cases}$
  - $\delta_2 = s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch}$,
  - $d_2 = \begin{cases} s.\texttt{delegator}(v,a).\texttt{next\_delta\_stake} & \text{if } \delta_2 \leq s.\texttt{epoch} \\ 0 & \text{if } \delta_2 > s.\texttt{epoch} \end{cases}$
- $s.\texttt{withdrawal\_stake}(v,a,s.\texttt{epoch}) = \sum_{i \in \texttt{uint8}} w_i$, where
  - $\tau_i = s.\texttt{withdrawal\_request}(v,a,i).\texttt{epoch}$,
  - $w_i = \begin{cases} s.\texttt{withdrawal\_request}(v,a,i).\texttt{amount} & \text{if } \tau_i > s.\texttt{epoch} \\ 0 & \text{if } \tau_i \leq s.\texttt{epoch} \end{cases}$
- For all $v \in \texttt{uint64}$,

$$\begin{aligned} s.\texttt{active\_consensus\_view}(v).\texttt{stake} = &\Big(\sum_{a \in \texttt{address}} s.\texttt{delegator\_stake}(v,a,s.\texttt{epoch}) \Big) \\ &+ \Big(\sum_{a \in \texttt{address}} s.\texttt{withdrawal\_stake}(v,a,s.\texttt{epoch}) \Big). \end{aligned}$$

3.4 Accumulated Reward per Token Invariants

In the model, each entry of `acc_rpt` carries two components: `value`, the accumulated reward per token itself, and `refcount`, a reference count. Where `acc_rpt` is used as a number elsewhere in this document, it refers to the `value` component.

- For all $t, v \in \texttt{uint64}$ with $v > 0$,

$$\begin{aligned} s.\texttt{acc\_rpt}(t,v).\texttt{refcount} = &\Big(\sum_{a \in \texttt{address}} I\big(s.\texttt{delegator}(v,a).\texttt{delta\_epoch},\ t\big)\Big) \\ &+ \Big(\sum_{a \in \texttt{address}} I\big(s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch},\ t\big)\Big) \\ &+ \Big(\sum_{\substack{a \in \texttt{address} \\ i \in \texttt{uint8}}} I\big(s.\texttt{withdrawal\_request}(v,a,i).\texttt{epoch},\ t\big)\Big), \end{aligned}$$

  where $I : \texttt{uint64}^2 \to \texttt{uint256}$,

$$I(x,y) = \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{if } x \neq y. \end{cases}$$

- For all $t > s.\texttt{epoch} + 2$ and $v \in \texttt{uint64}$, $s.\texttt{acc\_rpt}(t,v).\texttt{refcount} = 0$.
- $s.\texttt{acc\_rpt}(t,0).\texttt{refcount} = 0$ for all $t \in \texttt{uint64}$.
- For all $t, v \in \texttt{uint64}$, $s.\texttt{acc\_rpt}(t,v).\texttt{refcount} = 0$ implies $s.\texttt{acc\_rpt}(t,v).\texttt{value} = 0$.
- For $t_1, t_2, v \in \texttt{uint64}$ such that $t_1 \leq t_2$, $s.\texttt{acc\_rpt}(t_2,v).\texttt{refcount} \neq 0$ implies

$$s.\texttt{acc\_rpt}(t_1,v).\texttt{value} \leq s.\texttt{acc\_rpt}(t_2,v).\texttt{value}.$$

- For all $t, v \in \texttt{uint64}$,

$$s.\texttt{acc\_rpt}(t,v).\texttt{value} \leq s.\texttt{val\_execution}(v).\texttt{acc\_rpt}.$$

- For $v \in \texttt{uint64}$, $a \in \texttt{address}$, and $i \in \texttt{uint8}$,

$$s.\texttt{acc\_rpt}\big(s.\texttt{withdrawal\_request}(v,a,i).\texttt{epoch},\ v\big) \geq s.\texttt{withdrawal\_request}(v,a,i).\texttt{acc}.$$

- For $v \in \texttt{uint64}$ and $a \in \texttt{address}$,
  - $s.\texttt{delegator}(v,a).\texttt{acc\_rpt} \leq s.\texttt{val\_execution}(v).\texttt{acc\_rpt}$.
  - $0 < s.\texttt{delegator}(v,a).\texttt{delta\_epoch} \leq s.\texttt{epoch}$ implies

$$s.\texttt{delegator}(v,a).\texttt{acc\_rpt} \leq s.\texttt{acc\_rpt}\big(s.\texttt{delegator}(v,a).\texttt{delta\_epoch},\ v\big).$$

  - $0 < s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch} \leq s.\texttt{epoch}$ implies

$$s.\texttt{delegator}(v,a).\texttt{acc\_rpt} \leq s.\texttt{acc\_rpt}\big(s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch},\ v\big).$$

  - $0 < s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch} \leq s.\texttt{epoch}$ implies

$$\begin{aligned} &s.\texttt{acc\_rpt}\big(s.\texttt{delegator}(v,a).\texttt{delta\_epoch},\ v\big) \\ &\quad \leq s.\texttt{acc\_rpt}\big(s.\texttt{delegator}(v,a).\texttt{next\_delta\_epoch},\ v\big). \end{aligned}$$

3.5 Linked List Invariants

The following two invariants are required by the linked lists:

- $s.\texttt{last\_val\_id} < 2^{64} - 1$.
- $s.\texttt{delegator}(v, 2^{160} - 1).\texttt{stake} = 0$ for all $v \in \texttt{uint64}$.

The linked list invariants and later sections make use of the following linked list traversal sequences:

- $\texttt{delegator\_list}(v, s) = \big(A_0(v), A_1(v), \dots\big)$, for
  - $v \in \texttt{uint64}$,
  - $A_0(v) = s.\texttt{delegator}(v, 2^{160} - 1).\texttt{list\_node.anext}$,
  - $A_{i+1}(v) = \begin{cases} s.\texttt{delegator}(v, A_i(v)).\texttt{list\_node.anext} & \text{if } A_i(v) \neq 0 \\ 0 & \text{if } A_i(v) = 0 \end{cases}$
- $\texttt{validator\_list}(a, s) = \big(V_0(a), V_1(a), \dots\big)$, for
  - $a \in \texttt{address}$,
  - $V_0(a) = s.\texttt{delegator}(2^{64} - 1, a).\texttt{list\_node.inext}$,
  - $V_{i+1}(a) = \begin{cases} s.\texttt{delegator}(V_i(a), a).\texttt{list\_node.inext} & \text{if } V_i(a) \neq 0 \\ 0 & \text{if } V_i(a) = 0 \end{cases}$

It is an invariant that these sequences converge to 0:

- There exists an $M \geq 0$ such that $A_i(v) = 0$ for all $i \geq M$ and all $v \in \texttt{uint64}$.
- There exists an $N \geq 0$ such that $V_i(a) = 0$ for all $i \geq N$ and all $a \in \texttt{address}$.

The final linked list invariants are:

- The elements $\big(A_0(v), \dots, A_{M-1}(v)\big)$ are pairwise distinct, for all $v \in \texttt{uint64}$.
- The elements $\big(V_0(a), \dots, V_{N-1}(a)\big)$ are pairwise distinct, for all $a \in \texttt{address}$.
- v is in $\big(V_0(a), \dots, V_{N-1}(a)\big)$ if and only if a is in $\big(A_0(v), \dots, A_{M-1}(v)\big)$, for all v, a.
- a is in $\big(A_0(v), \dots, A_{M-1}(v)\big)$ if and only if

$$s.\texttt{delegator}(v, a).\texttt{stake} + s.\texttt{delegator}(v, a).\texttt{delta\_stake} + s.\texttt{delegator}(v, a).\texttt{next\_delta\_stake} \neq 0,$$

  for all v, a.

3.6 Solvency Invariants

The first solvency invariant expresses that stake and rewards are exactly accounted for by the balance of the staking contract:

$$s.\texttt{balance\_of}(\texttt{STAKING\_CA}) = \Big(\sum_{v \in \texttt{uint64}} s.\texttt{val\_execution}(v).\texttt{stake}\Big)$$

$$
\begin{aligned}
&+ \Big(\sum_{v \in \texttt{uint64}} s.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} \Big) \\
&+ \Big(\sum_{\substack{v \in \texttt{uint64} \\ a \in \texttt{address}}} s.\texttt{delegator}(v,a).\texttt{rewards} \Big) \\
&+ \Big(\sum_{\substack{v \in \texttt{uint64} \\ a \in \texttt{address} \\ i \in \texttt{uint8}}} s.\texttt{withdrawal\_request}(v,a,i).\texttt{amount} \Big).
\end{aligned}
$$

Moreover, all pending rewards are accounted for, modulo rounding errors: for all $v \in \texttt{uint64}$,

$$
\begin{aligned}
\sum_{a \in \texttt{address}} \texttt{pending\_rewards}(v,a,s) &\leq s.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} \\
&\leq \beta + \sum_{a \in \texttt{address}} \texttt{pending\_rewards}(v,a,s),
\end{aligned}
$$

where $\beta \in \texttt{uint256}$ is a rounding error upper bound, defined by

$$
\begin{aligned}
\beta = {} & s.\texttt{error\_bound} \\
&+ \Big(\sum_{v \in \texttt{uint64},\ a \in \texttt{address},\ i \in \texttt{uint8}} I\big(s.\texttt{withdrawal\_request}(v,a,i).\texttt{amount}\big) \Big) \\
&+ \Big(\sum_{v \in \texttt{uint64},\ a \in \texttt{address}} 3 \cdot I\big(d.\texttt{stake} + d.\texttt{delta\_stake} + d.\texttt{next\_delta\_stake}\big) \Big),
\end{aligned}
$$

with $d = s.\texttt{delegator}(v,a)$ in the last sum, and using

$$
I(x) = \begin{cases} 1 & \text{if } x \neq 0, \\ 0 & \text{if } x = 0. \end{cases}
$$

The last solvency invariant states that a validator's stake is the same as the delegated stake:

$$
\begin{aligned}
s.\texttt{val\_execution}(v).\texttt{stake} = {} & \Big(\sum_{a \in \texttt{address}} s.\texttt{delegator}(v,a).\texttt{stake} \Big) \\
&+ \Big(\sum_{a \in \texttt{address}} s.\texttt{delegator}(v,a).\texttt{delta\_stake} \Big) \\
&+ \Big(\sum_{a \in \texttt{address}} s.\texttt{delegator}(v,a).\texttt{next\_delta\_stake} \Big),
\end{aligned}
$$

for all $v \in \texttt{uint64}$.

4 Function Specifications

This section specifies some of the pre-conditions and post-conditions of the partial interface functions. The pre-conditions define the subset of the domain (inputs) on which the partial interface functions are defined. The post-conditions define a subset of the codomain (outputs)

within which the results of the partial interface functions lie. Throughout, $\tilde{s}$ denotes the resulting (output) state.

All the staking contract partial interface functions have pre-conditions and post-conditions related to the structure of their inputs and outputs. These are implicitly accounted for in the model by restricting the interface functions to specific domains and codomains (input and output sets).

It is important to note that the specifications below include the transfer of MON balances involved in calling `payable` staking contract interface functions, which is needed by the solvency invariants. This transfer is not part of the staking contract itself; it is instead handled by Monad before entering the staking contract.

4.1 Specification of `precompile_add_validator`

$$\texttt{precompile\_add\_validator} : \texttt{bytes} \times \texttt{bytes} \times \texttt{bytes} \times \texttt{address} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{uint64} \times \texttt{State}$$

The result of $\texttt{precompile\_add\_validator}(m, b, c, \alpha, \mu, s)$ is defined if the following pre-conditions are satisfied:

- m can be decoded into the following structure (decoding details are given in the source code):

$$(b_0, c_0, a, x, p) \in \texttt{bytes} \times \texttt{bytes} \times \texttt{address} \times \texttt{uint256} \times \texttt{uint256}$$

- $s.\texttt{balance\_of}(\texttt{STAKING\_CA}) + \mu \geq \mu$ (no overflow)
- $s.\texttt{last\_val\_id} + 1 \geq 1$ (no overflow)
- $x = \mu$
- $\mu \leq \texttt{UNIT\_BIAS}$ (this is a pre-condition because of error-bound tracking; it is not part of the C++ code)
- $\mu \geq \texttt{MIN\_VALIDATE\_STAKE}$
- $\mu \geq \texttt{DUST\_THRESHOLD}$
- b has the structure of a SECP signature
- b_0 has the structure of a SECP public key
- the signature b verifies that b_0 has signed the message m
- c has the structure of a BLS signature
- c_0 has the structure of a BLS public key
- the signature c verifies that c_0 has signed the message m
- $p \leq \texttt{MAX\_COMMISSION}$
- $s.\texttt{val\_id}(y) = 0$, for y the SECP address from b_0
- $s.\texttt{val\_id\_bls}(z) = 0$, for z the BLS address from c_0

The post-conditions on

$$(v, \tilde{s}) = \texttt{precompile\_add\_validator}(m, b, c, \alpha, \mu, s)$$

are:

- $\tilde{s}.\texttt{balance\_of}(\texttt{STAKING\_CA}) = s.\texttt{balance\_of}(\texttt{STAKING\_CA}) + \mu$
- $\tilde{s}.\texttt{last\_val\_id} = v = s.\texttt{last\_val\_id} + 1$
- $\tilde{s}.\texttt{val\_execution}(v).\texttt{keys.secp\_pubkey} = b_0$
- $\tilde{s}.\texttt{val\_execution}(v).\texttt{keys.bls\_pubkey} = c_0$
- $\tilde{s}.\texttt{val\_execution}(v).\texttt{commission} = p$
- $\tilde{s}.\texttt{val\_execution}(v).\texttt{stake} = \mu$
- $s.\texttt{in\_epoch\_delay\_period} = 1$ implies $\tilde{s}.\texttt{delegator\_stake}(v, a, e) = \mu$ for all $e \geq s.\texttt{epoch} + 2$
- $s.\texttt{in\_epoch\_delay\_period} = 0$ implies $\tilde{s}.\texttt{delegator\_stake}(v, a, e) = \mu$ for all $e \geq s.\texttt{epoch} + 1$
- v is an element of $\tilde{s}.\texttt{valset\_execution}$ implies $\mu \geq \texttt{ACTIVE\_VALIDATOR\_STAKE}$
- a is an element of $\texttt{delegator\_list}(v, \tilde{s})$

4.2 Specification of `precompile_delegate`

$$\texttt{precompile\_delegate} : \texttt{uint64} \times \texttt{address} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{bool} \times \texttt{State}$$

The pre-conditions of $\texttt{precompile\_delegate}(v, \alpha, \mu, s)$ are:

- $s.\texttt{balance\_of}(\texttt{STAKING\_CA}) + \mu \geq \mu$ (no overflow)
- $\mu = 0$ or $\mu \geq \texttt{DUST\_THRESHOLD}$
- $\mu \neq 0$ implies $s.\texttt{val\_execution}(v) \neq 0$
- $s.\texttt{val\_execution}(v).\texttt{stake} + \mu \leq \texttt{UNIT\_BIAS}$ (this is a pre-condition because of error-bound tracking)

The post-conditions of $(r, \tilde{s}) = \texttt{precompile\_delegate}(v, \alpha, \mu, s)$ are:

- $r = 1$,

and if $\mu \neq 0$, the following post-conditions also hold:

- $\tilde{s}.\texttt{balance\_of}(\texttt{STAKING\_CA}) = s.\texttt{balance\_of}(\texttt{STAKING\_CA}) + \mu$
- $s.\texttt{in\_epoch\_delay\_period} = 1$ implies $\tilde{s}.\texttt{delegator\_stake}(v, \alpha, e) = s.\texttt{delegator\_stake}(v, \alpha, e) + \mu$ for all $e \geq s.\texttt{epoch} + 2$
- $s.\texttt{in\_epoch\_delay\_period} = 0$ implies $\tilde{s}.\texttt{delegator\_stake}(v, \alpha, e) = s.\texttt{delegator\_stake}(v, \alpha, e) + \mu$ for all $e \geq s.\texttt{epoch} + 1$

- $\tilde{s}.\texttt{val\_execution}(v).\texttt{stake} = s.\texttt{val\_execution}(v).\texttt{stake} + \mu$
- α is an element of $\texttt{delegator\_list}(v, \tilde{s})$
- $\tilde{s}.\texttt{error\_bound} = s.\texttt{error\_bound} + 3$

4.3 Specification of `precompile_undelegate`

$$\texttt{precompile\_undelegate} : \texttt{uint64} \times \texttt{uint256} \times \texttt{uint8} \times \texttt{address} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{bool} \times \texttt{State}$$

The pre-conditions of $\texttt{precompile\_undelegate}(v, d, i, \alpha, \mu, s)$ are:

- $\mu = 0$
- $d \leq \sigma_1 + \sigma_2 + \sigma_3$
- $d > 0$ implies
  - $s.\texttt{withdrawal\_request}(v, \alpha, i).\texttt{amount} = 0$ and
  - $s.\texttt{epoch} + \texttt{WITHDRAWAL\_DELAY} + 2 \geq s.\texttt{epoch}$ (no overflow)

where

- $\sigma_1 = s.\texttt{delegator}(v, \alpha).\texttt{stake}$,
- $\sigma_2 = \begin{cases} s.\texttt{delegator}(v, \alpha).\texttt{delta\_stake} & \text{if } s.\texttt{epoch} \leq \delta \\ 0 & \text{if } s.\texttt{epoch} > \delta \end{cases}$
  for $\delta = s.\texttt{delegator}(v, \alpha).\texttt{delta\_epoch}$,
- $\sigma_3 = \begin{cases} s.\texttt{delegator}(v, \alpha).\texttt{next\_delta\_stake} & \text{if } s.\texttt{epoch} \leq \delta \\ 0 & \text{if } s.\texttt{epoch} > \delta \end{cases}$
  for $\delta = s.\texttt{delegator}(v, \alpha).\texttt{next\_delta\_epoch}$.

The post-conditions of $(r, \tilde{s}) = \texttt{precompile\_undelegate}(v, d, i, \alpha, \mu, s)$ are:

- $r = 1$,

and if $d > 0$, also the post-conditions:

- $\tilde{s}.\texttt{withdrawal\_request}(v, \alpha, i).\texttt{amount} = \tilde{d}$
- $s.\texttt{in\_epoch\_delay\_period} = 1$ implies $\tilde{s}.\texttt{withdrawal\_request}(v, \alpha, i).\texttt{epoch} = s.\texttt{epoch} + 2$
- $s.\texttt{in\_epoch\_delay\_period} = 0$ implies $\tilde{s}.\texttt{withdrawal\_request}(v, \alpha, i).\texttt{epoch} = s.\texttt{epoch} + 1$
- For $s.\texttt{epoch} \leq e < \tilde{s}.\texttt{withdrawal\_request}(v, \alpha, i).\texttt{epoch}$,

$$\tilde{s}.\texttt{withdrawal\_stake}(v, \alpha, e) = s.\texttt{withdrawal\_stake}(v, \alpha, e) + \tilde{d}$$

- $\tilde{s}.\texttt{val\_execution}(v).\texttt{stake} = s.\texttt{val\_execution}(v).\texttt{stake} - \tilde{d}$

- If both $\tilde{s}.\texttt{val\_execution}(v).\texttt{stake} \geq \texttt{ACTIVE\_VALIDATOR\_STAKE}$ and

$$s.\texttt{delegator}(v,\alpha).\texttt{stake} + s.\texttt{delegator}(v,\alpha).\texttt{delta\_stake} + s.\texttt{delegator}(v,\alpha).\texttt{next\_delta\_stake} \geq \texttt{MIN\_VALIDATE\_STAKE},$$

  then v is an element of $\tilde{s}.\texttt{valset\_execution}$
- For all $e \geq s.\texttt{epoch}$, $\tilde{s}.\texttt{delegator\_stake}(v,\alpha,e) = s.\texttt{delegator\_stake}(v,\alpha,e) - \tilde{d}$
- $\sigma_1 = \sigma_2 = \sigma_3 = 0$ implies that α is *not* in $\texttt{delegator\_list}(v,\tilde{s})$
- $\tilde{s}.\texttt{error\_bound} = s.\texttt{error\_bound} + 3$

where

$$\tilde{d} = \begin{cases} d & \text{if } x - d \geq \texttt{DUST\_THRESHOLD} \\ x & \text{if } x - d < \texttt{DUST\_THRESHOLD} \end{cases} \qquad \text{for } x = \sigma_1 + \sigma_2 + \sigma_3.$$

4.4 Specification of `precompile_compound`

$$\texttt{precompile\_compound} : \texttt{uint64} \times \texttt{address} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{bool} \times \texttt{State}$$

The pre-conditions of $\texttt{precompile\_compound}(v,\alpha,\mu,s)$ are:

- $\mu = 0$
- $x = 0$ or $x \geq \texttt{DUST\_THRESHOLD}$
- $s.\texttt{val\_execution}(v).\texttt{stake} + x \leq \texttt{UNIT\_BIAS}$ (this is a pre-condition because of error-bound tracking)

where

$$x = s.\texttt{delegator}(v,\alpha).\texttt{rewards} + \texttt{unaccumulated\_rewards}(v,\alpha,s).$$

The post-conditions of $(r,\tilde{s}) = \texttt{precompile\_compound}(v,\alpha,\mu,s)$ are:

- $r = 1$,

and if $x > 0$, the additional post-conditions:

- $s.\texttt{in\_epoch\_delay\_period} = 1$ implies $\tilde{s}.\texttt{delegator\_stake}(v,\alpha,e) = s.\texttt{delegator\_stake}(v,\alpha,e) + x$ for all $e \geq s.\texttt{epoch} + 2$
- $s.\texttt{in\_epoch\_delay\_period} = 0$ implies $\tilde{s}.\texttt{delegator\_stake}(v,\alpha,e) = s.\texttt{delegator\_stake}(v,\alpha,e) + x$ for all $e \geq s.\texttt{epoch} + 1$
- $\tilde{s}.\texttt{val\_execution}(v).\texttt{stake} = s.\texttt{val\_execution}(v).\texttt{stake} + x$
- If both $\tilde{s}.\texttt{val\_execution}(v).\texttt{stake} \geq \texttt{ACTIVE\_VALIDATOR\_STAKE}$ and

$$s.\texttt{delegator}(v,\alpha).\texttt{stake} + s.\texttt{delegator}(v,\alpha).\texttt{delta\_stake} + s.\texttt{delegator}(v,\alpha).\texttt{next\_delta\_stake} \geq \texttt{MIN\_VALIDATE\_STAKE},$$

  then v is an element of $\tilde{s}.\texttt{valset\_execution}$

- $\tilde{s}.\texttt{error\_bound} = s.\texttt{error\_bound} + 3$
- $\tilde{s}.\texttt{unit\_bias\_rewards}(v, \alpha) = s.\texttt{unit\_bias\_rewards}(v, \alpha) - x \cdot \texttt{UNIT\_BIAS}$

4.5 Specification of `precompile_withdraw`

$\texttt{precompile\_withdraw} : \texttt{uint64} \times \texttt{uint8} \times \texttt{address} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{bool} \times \texttt{State}$

The pre-conditions of $\texttt{precompile\_withdraw}(v, i, \alpha, \mu, s)$ are:

- $\mu = 0$
- $s.\texttt{withdrawal\_request}(v, \alpha, i).\texttt{amount} > 0$
- $s.\texttt{epoch} + \texttt{WITHDRAWAL\_DELAY} \geq \texttt{WITHDRAWAL\_DELAY}$ (no overflow)
- $s.\texttt{withdrawal\_request}(v, \alpha, i).\texttt{epoch} + \texttt{WITHDRAWAL\_DELAY} \leq s.\texttt{epoch}$
- $s.\texttt{balance\_of}(\alpha) + y \geq y$ (no overflow)

where

$$x = \texttt{withdrawal\_reward}(v, \alpha, i, s),$$
$$y = s.\texttt{withdrawal\_request}(v, \alpha, i).\texttt{amount} + x.$$

The post-conditions of $(r, \tilde{s}) = \texttt{precompile\_withdraw}(v, i, \alpha, \mu, s)$ are:

- $\tilde{s}.\texttt{balance\_of}(\texttt{STAKING\_CA}) = s.\texttt{balance\_of}(\texttt{STAKING\_CA}) - y$
- $\tilde{s}.\texttt{balance\_of}(\alpha) = s.\texttt{balance\_of}(\alpha) + y$
- $\tilde{s}.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} = s.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} - x$
- $\tilde{s}.\texttt{error\_bound} = s.\texttt{error\_bound} + 1$
- $\tilde{s}.\texttt{unit\_bias\_rewards}(v, \alpha) = s.\texttt{unit\_bias\_rewards}(v, \alpha) - x \cdot \texttt{UNIT\_BIAS}$

4.6 Specification of `precompile_change_commission`

$\texttt{precompile\_change\_commission} : \texttt{uint64} \times \texttt{uint256} \times \texttt{address} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{bool} \times \texttt{State}$

The pre-conditions of $\texttt{precompile\_change\_commission}(v, p, \alpha, \mu, s)$ are:

- $\mu = 0$
- $\alpha = s.\texttt{val\_execution}(v).\texttt{auth\_address}$
- $p \leq \texttt{MAX\_COMMISSION}$

The post-conditions of $(r, \tilde{s}) = \texttt{precompile\_change\_commission}(v, p, \alpha, \mu, s)$ are:

- $r = 1$
- $\tilde{s}.\texttt{val\_execution}(v).\texttt{commission} = p$

4.7 Specification of `precompile_claim_rewards`

$\texttt{precompile\_claim\_rewards} : \texttt{uint64} \times \texttt{address} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{bool} \times \texttt{State}$

The pre-conditions of $\texttt{precompile\_claim\_rewards}(v, \alpha, \mu, s)$ are:

- $\mu = 0$

The post-conditions of $(r, \tilde{s}) = \texttt{precompile\_claim\_rewards}(v, \alpha, \mu, s)$ are:

- $r = 1$
- $\tilde{s}.\texttt{balance\_of}(\alpha) = s.\texttt{balance\_of}(\alpha) + x$
- $\tilde{s}.\texttt{balance\_of}(\texttt{STAKING\_CA}) = s.\texttt{balance\_of}(\texttt{STAKING\_CA}) - x$
- $\tilde{s}.\texttt{error\_bound} = s.\texttt{error\_bound} + 3$
- $\tilde{s}.\texttt{unit\_bias\_rewards}(v, \alpha) = s.\texttt{unit\_bias\_rewards}(v, \alpha) - x \cdot \texttt{UNIT\_BIAS}$

where

$$x = s.\texttt{delegator}(v, \alpha).\texttt{rewards} + \texttt{unaccumulated\_rewards}(v, \alpha, s).$$

4.8 Specification of `precompile_external_reward`

$\texttt{precompile\_external\_reward} : \texttt{uint64} \times \texttt{address} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{bool} \times \texttt{State}$

The pre-conditions of $\texttt{precompile\_external\_reward}(v, \alpha, \mu, s)$ are:

- $s.\texttt{active\_consensus\_view}(v).\texttt{stake} \neq 0$
- $\texttt{MIN\_EXTERNAL\_REWARD} \leq \mu \leq \texttt{MAX\_EXTERNAL\_REWARD}$

The post-conditions of $(r, \tilde{s}) = \texttt{precompile\_external\_reward}(v, \alpha, \mu, s)$ are:

- $r = 1$
- $\tilde{s}.\texttt{error\_bound} = s.\texttt{error\_bound} + 1$
- For all $a \in \texttt{address}$, $\tilde{s}.\texttt{unit\_bias\_rewards}(v, a) = s.\texttt{unit\_bias\_rewards}(v, a) + R \cdot D(a)$
- $\tilde{s}.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} = s.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} + \mu$

where

- $R = (\mu \cdot \texttt{UNIT\_BIAS}) / s.\texttt{active\_consensus\_view}(v).\texttt{stake}$ and
- $D(a) = s.\texttt{delegator\_stake}(v, a, s.\texttt{epoch}) + s.\texttt{withdrawal\_stake}(v, a, s.\texttt{epoch})$.

4.9 Specification of `syscall_reward`

$$\texttt{syscall\_reward} : \texttt{uint64} \times \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{State}$$

The pre-conditions of $\texttt{syscall\_reward}(v, \rho, s)$ are:

- $s.\texttt{active\_consensus\_view}(v).\texttt{stake} \neq 0$

The post-conditions of $\tilde{s} = \texttt{syscall\_reward}(v, \rho, s)$ are:

- $\tilde{s}.\texttt{error\_bound} = s.\texttt{error\_bound} + 1$
- $\tilde{s}.\texttt{unit\_bias\_rewards}(v, A) = s.\texttt{unit\_bias\_rewards}(v, A) + c \cdot \texttt{UNIT\_BIAS} + R \cdot D(A)$
- For all $a \in \texttt{address}$ such that $a \neq A$, $\tilde{s}.\texttt{unit\_bias\_rewards}(v, a) = s.\texttt{unit\_bias\_rewards}(v, a) + R \cdot D(a)$
- $\tilde{s}.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} = s.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} + \rho - c$
- $\tilde{s}.\texttt{proposer\_val\_id} = v$

where

- $A = s.\texttt{val\_execution}(v).\texttt{auth\_address}$,
- $c = (\rho \cdot s.\texttt{active\_consensus\_view}(v).\texttt{commission})/\texttt{MON}$,
- $R = ((\rho - c) \cdot \texttt{UNIT\_BIAS})/s.\texttt{active\_consensus\_view}(v).\texttt{stake}$, and
- $D(a) = s.\texttt{delegator\_stake}(v, a, s.\texttt{epoch}) + s.\texttt{withdrawal\_stake}(v, a, s.\texttt{epoch})$.

4.10 Specification of `syscall_snapshot`

$$\texttt{syscall\_snapshot} : \texttt{State} \rightharpoonup \texttt{State}$$

The pre-conditions of $\texttt{syscall\_snapshot}(s)$ are:

- $s.\texttt{in\_epoch\_delay\_period} = 0$
- $s.\texttt{epoch} \neq 0$ (not part of the C++ code)

The post-conditions of $\tilde{s} = \texttt{syscall\_snapshot}(s)$ are:

- $\tilde{s}.\texttt{in\_epoch\_delay\_period} = 1$
- For all $v \in \texttt{uint64}$, v an element of $\tilde{s}.\texttt{valset\_execution}$ implies $\tilde{s}.\texttt{val\_execution}(v).\texttt{stake} \geq \texttt{ACTIVE\_VALIDATOR\_STAKE}$
- For all $v, u \in \texttt{uint64}$, if v is in $s.\texttt{valset\_consensus}$ and u is not in $s.\texttt{valset\_consensus}$, then either
  - $s.\texttt{val\_execution}(v).\texttt{stake} \geq s.\texttt{val\_execution}(u).\texttt{stake}$, or

– the inequality

$$s.\texttt{delegator}(u, A).\texttt{stake} + s.\texttt{delegator}(u, A).\texttt{delta\_stake} + s.\texttt{delegator}(u, A).\texttt{next\_delta\_stake} < \texttt{MIN\_VALIDATE\_STAKE}$$

holds, for $A = s.\texttt{val\_execution}(u).\texttt{auth\_address}$.

4.11 Specification of `syscall_on_epoch_change`

$$\texttt{syscall\_on\_epoch\_change} : \texttt{uint64} \times \texttt{State} \rightharpoonup \texttt{State}$$

The pre-conditions of $\texttt{syscall\_on\_epoch\_change}(e, s)$ are:

- $e > s.\texttt{epoch}$
- The following pre-condition is not part of the C++ code: if $s.\texttt{epoch} \neq 0$ (if bootstrapped), then
  - $e = s.\texttt{epoch} + 1$ and
  - $s.\texttt{in\_epoch\_delay\_period} = 1$.

The post-conditions of $\tilde{s} = \texttt{syscall\_on\_epoch\_change}(e, s)$ are:

- $\tilde{s}.\texttt{in\_epoch\_delay\_period} = 0$
- $\tilde{s}.\texttt{epoch} = e$
- For all $v \in \texttt{uint64}$, $\tilde{s}.\texttt{active\_consensus\_view}(v) = \tilde{s}.\texttt{consensus\_view}(v)$

4.12 Specification of `precompile_get_delegator`

The `precompile_get_delegator` function increases `error_bound` by 3 and increases the delegator's `rewards` field by `unaccumulated_rewards`, after which `unaccumulated_rewards` becomes zero.

4.13 Specification of `distribute_priority_fees`

$$\texttt{distribute\_priority\_fees} : \texttt{uint256} \times \texttt{State} \rightharpoonup \texttt{State}$$

The pre-conditions of $\texttt{distribute\_priority\_fees}(\rho, s)$ are:

- $v \neq 0$
- $s.\texttt{active\_consensus\_view}(v).\texttt{stake} \neq 0$

where

- $v = s.\texttt{proposer\_val\_id}$,

- $c = (\rho \cdot s.\texttt{active\_consensus\_view}(v).\texttt{commission})/\texttt{MON}$,
- $R = ((\rho - c) \cdot \texttt{UNIT\_BIAS})/s.\texttt{active\_consensus\_view}(v).\texttt{stake}$,
- $D(a) = s.\texttt{delegator\_stake}(v, a, s.\texttt{epoch}) + s.\texttt{withdrawal\_stake}(v, a, s.\texttt{epoch})$, and
- $A = s.\texttt{val\_execution}(v).\texttt{auth\_address}$.

The post-conditions of $\tilde{s} = \texttt{distribute\_priority\_fees}(\rho, s)$ are:

- If $\rho - c \geq \texttt{MIN\_EXTERNAL\_REWARD}$, then
  - $\tilde{s}.\texttt{balance\_of}(\texttt{STAKING\_CA}) = s.\texttt{balance\_of}(\texttt{STAKING\_CA}) + \rho$,
  - $\tilde{s}.\texttt{error\_bound} = s.\texttt{error\_bound} + 1$,
  - $\tilde{s}.\texttt{unit\_bias\_rewards}(v, A) =$ $s.\texttt{unit\_bias\_rewards}(v, A) + c \cdot \texttt{UNIT\_BIAS} + R \cdot D(A)$,
  - for all $a \in \texttt{address}$ such that $a \neq A$, $\tilde{s}.\texttt{unit\_bias\_rewards}(v, a) =$ $s.\texttt{unit\_bias\_rewards}(v, a) + R \cdot D(a)$, and
  - $\tilde{s}.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} =$ $s.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} + \rho - c$.
- Otherwise, if $\rho - c < \texttt{MIN\_EXTERNAL\_REWARD}$, then
  - $\tilde{s}.\texttt{balance\_of}(\texttt{STAKING\_CA}) = s.\texttt{balance\_of}(\texttt{STAKING\_CA}) + c$,
  - $\tilde{s}.\texttt{error\_bound} = s.\texttt{error\_bound}$,
  - $\tilde{s}.\texttt{unit\_bias\_rewards}(v, A) =$ $s.\texttt{unit\_bias\_rewards}(v, A) + c \cdot \texttt{UNIT\_BIAS}$,
  - $\tilde{s}.\texttt{unit\_bias\_rewards}(v, a) = s.\texttt{unit\_bias\_rewards}(v, a)$ for all $a \neq A$, and
  - $\tilde{s}.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards} =$ $s.\texttt{val\_execution}(v).\texttt{unclaimed\_rewards}$.