Hydra HeadV2 Specification: Coordinated Head protocol

DRAFT

Sebastian Nagel
sebastian.nagel@iohk.io

Sasha Bogicevic
sasha.bogicevic@iohk.io

Franco Testagrossa
franco.testagrossa@iohk.io

Daniel Firth
daniel.firth@iohk.io

Noon van der Silk
noon.vandersilk@iohk.io

Veronika Romashkina
veronika.romashkina@iohk.io

Contents

1 Introduction

This document specifies the 'Coordinated Hydra Head' protocol to be implemented as the second version of Hydra Head on Cardano - **Hydra HeadV2**. The protocol is derived from variants described in the original paper [1], but was further simplified to make an implementation on Cardano possible, efficient, and user friendly.

Note that the format and scope of this document is (currently) also inspired by the paper and hence does not include a definition of the networking protocols or concrete message formats. **TODO:** Add: network specification (message formats) It is structured similarly, but focuses on a single variant, and avoids indirections and unnecessary generalizations. The document is kept in sync with the reference implementation available on Github [2].

First, a high-level overview of the protocol and how it differs from legacy variants of the Head protocol is given in Section 2. Relevant definitions and notations are introduced in Section 3, while Section 4 describes protocol setup and assumptions. Then, the actual on-chain transactions of the protocol are defined in Section 5, before the off-chain protocol part specifies behavior of Hydra parties off-chain and ties the knot with on-chain transactions in Section 6. At last, Section 7 gives the security definition, properties and proofs for the Coordinated Head protocol.

2 Protocol Overview

The Hydra Head protocol provides functionality to lock a set of UTxOs on a blockchain, referred to as the *mainchain*, and evolve it inside a so-called off-chain *head*, independently of the mainchain. At any point, the head can be closed with the effect that the locked set of UTxOs on the mainchain is replaced by the latest set of UTxOs inside the head. The protocol guarantees full wealth preservation: no generation of funds can happen off-chain (inside a head) and no responsive honest party involved in a head can ever lose any funds other than by consenting to give them away. In exchange for decreased liveness guarantees (stop any time), it can essentially proceed at network speed under good conditions, thereby reducing latency and increasing throughput. At the same time, the head protocol provides the same capabilities as the mainchain by reusing the same ledger model and transaction formats — making the protocol "isomorphic".

2.1 Opening the head

To create a head-protocol instance, any party may take the role of an *initiator* and ask other parties, the *head members*, to participate in the head by exchanging public keys and agreeing on other protocol parameters. These public keys are used for both, the authentication of head-related on-chain transactions that are restricted to head members (e.g., a non-member is not allowed to close the head) and for signing off-chain transactions in the head.

The initiator then establishes and directly opens the head by submitting an *init* transaction to the mainchain that contains the Hydra protocol parameters and mints special *participation tokens (PT)* identifying the head members. The *init* transaction initializes a state machine (see Figure 1) that manages the life-cycle of UTxOs in the head. The state machine comprises the four states: open, closed, fanoutProgress, and final. A *state thread token (ST)* minted in *init* marks the head output and ensures contract continuity [3].

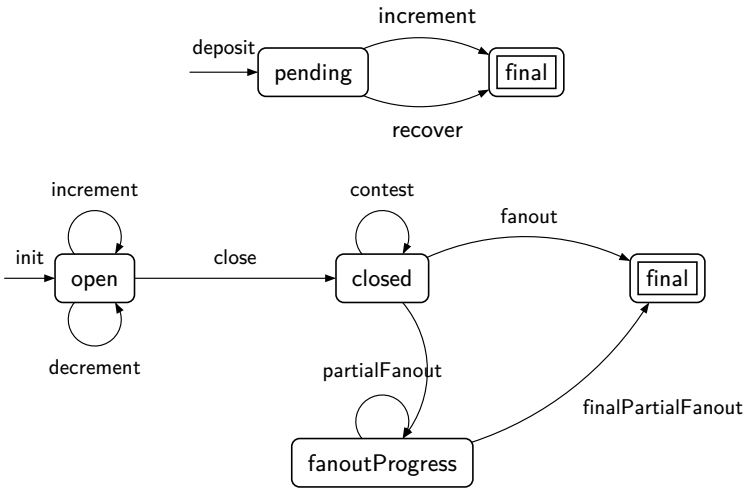

Figure 1: Maintain state diagram for this version of the Hydra protocol

The head is opened with an empty UTxO set. Participants can then add funds to the head by creating *deposit* transactions on the mainchain, which are subsequently incorporated into the head via *increment* transactions (see Section 5.2 and Section 5.4). Once in the open state, the head members start running the off-chain head protocol, which evolves the UTxO set independently of the mainchain.

2.2 The Coordinated Head protocol

The actual Head protocol starts once the head is open on-chain, with an initially empty UTxO set.

The protocol processes off-chain transactions by distributing them between participants, while each party maintains their view of the local UTxO state. That is, the current set of UTxOs evolved from the empty initial state by applying transactions as they are received from the other parties.

To confirm transactions and allow for distribution of the resulting UTxO set without needing the whole transaction history, snapshots are created by the protocol participants. The initial snapshot U_0 corresponds to the initial (empty) UTxO set, while snapshots thereafter $U_1, U_2, ...$ are created with monotonically increasing snapshot numbers.

For this, the next snapshot leader (round-robin) requests their view of a new confirmed state to be signed by all participants as a new snapshot. The leader does not need to send their local state, but only indicate, by hashes, the set of transactions to be included in order to obtain the to-be-snapshotted UTxO set.

The other participants sign the snapshot as soon as they have (also) seen the transactions that are to be processed on top of its preceding snapshot: a party's local state is always ahead of the latest confirmed snapshot.

Signatures are broadcast and aggregated by each party. When all signature parts of the multi-signature are received and verified, a snapshot is considered confirmed. As a consequence, a participant can safely delete (if wished) all transactions that have been processed into it as the snapshot's multi-signature is now evidence that this state once existed during the head evolution.

2.2.1 Updating an open head

Besides processing "normal" transactions, participants can also request to withdraw some UTxO they can spend from the Head and make it available on main chain via a *decrement* Section 5.5 transaction — the overall process is also called "decommit".

To add UTxO to an open head, anyone may create a deposit of one or more UTxO using a *deposit* Section 5.2 transaction. The head participants will observe this deposit and, once settled, request off-chain agreement to include the deposited UTxO in the form of a snapshot. With that agreement, an *increment* Section 5.4 transaction can be created and used to update the head state on the mainchain. A deadline is associated with the deposit, which ensures that the UTxO is locked up long enough to be safely consumed into the head without risk of double spending. Should a deposit have not been picked up in time, a *recover* Section 5.3 transaction allows anyone to unlock the original UTxO.

2.3 Closing the head

The head protocol is designed to allow any head member at any point in time to produce, without interaction, a certificate to close the head. This certificate is created from the latest confirmed, multi-signed snapshot. Using this certificate, the head member may "force close" the head by advancing the mainchain state machine to the closed state.

Once in closed, the state machine grants parties a contestation period, during which parties may contest the closure by posting the certificate of a newer snapshot on-chain in a contest transaction. Contesting leads back to the state closed and each party can contest at most once. After the contestation period has elapsed, the state machine may proceed to the final state. The state machine enforces that the outputs of the transaction leading to final correspond exactly to the latest UTxO set seen during the contestation period. **TODO:** Update to capture details about partial fanout.

2.4 Differences

In the Coordinated Head protocol, off-chain consensus is simplified by not having transactions confirmed concurrently to the snapshots (and to each other) but having the snapshot leader propose, in their snapshot, a set of transactions for explicit confirmation. The parties' views of confirmed transactions thus progress in sync with each other (once per confirmed snapshot), thus simplifying the close/contest procedure on the mainchain. Also, there is no need for conflict resolution as in Appendix B of [1]. In summary, the differences to the original Head protocol in [1] are:

- No hanging transactions due to 'coordination'.
- No acknowledgement nor confirmation of transactions.
- No confirmation message for snapshots (two-round local confirmation).
- Allow for deposits and decommits while head is open.
- No initialization phase: head opens directly with empty UTxO, funds added via deposits.

3 Preliminaries

This section introduces notation and other preliminaries used in the remainder of the specification.

3.1 Notation

The specification uses set-notation based approach while also inspired by [4] and [3]. Values a are in a set $a \in \mathcal{A}$, also indicated as being of some type $a : \mathcal{A}$, and multidimensional values are tuples drawn from a $\times$ product of multiple sets, e.g. $(a, b) \in (\mathcal{A} \times \mathcal{B})$. An empty set is indicated by $\emptyset$ and sets may be enumerated using $\{a_1 ... a_n\}$ notation. The $=$ operator means equality and $\leftarrow$ is explicit assignment of a variable or value to one or more variables. Projection is used to access the elements of a tuple, e.g. $(a, b)^{\downarrow 1} = a$. Functions are morphisms mapping from one set to another $x : \mathcal{A} \to f(x) : \mathcal{B}$, where function application of a function f to an argument x is written as $f(x)$.

Furthermore, given a set $\mathcal{A}$, let

- $\mathcal{A}^n$ be the set of all n-sized sequences over $\mathcal{A}$, and
- $\mathcal{A}^* = \bigcup_{i=0}^{n \in \mathbb{N}} \mathcal{A}^i$ be the set of all sequences over $\mathcal{A}$.

With this, we further define:

- $\perp$ indicates the absence of a value, e.g. an optional or not-yet-set variable holds $\perp$
- $\mathbb{B} = \{\texttt{false}, \texttt{true}\}$ are boolean values
- $\mathbb{N}$ are natural numbers $\{0, 1, 2, ...\}$
- $\mathbb{Z}$ are integer numbers $\{..., -2, -1, 0, 1, 2, ...\}$
- $\mathbb{H} = \bigcup_{n=0}^{\infty} \{0, 1\}^{8n}$ denotes a arbitrary string of bytes
- $\mathsf{concat} : \mathbb{H}^* \to \mathbb{H}$ is concatenating bytes, we also use operator $\bigoplus$ for this
- $\mathsf{hash} : x \to \mathbb{H}$ denotes a collision-resistant hashing function and $x^{\#}$ indicates the hash of x
- $\mathsf{bytes} : x \to \mathbb{H}$ denotes an invertible serialisation function mapping arbitrary data to bytes
- $a \parallel b = \mathsf{concat}(\mathsf{bytes}(a), \mathsf{bytes}(b))$ is an operator which concatenates the $\mathsf{bytes}(b)$ to the $\mathsf{bytes}(a)$
- Lists of values $l \in \mathcal{A}^*$ are written as $l = [x_1, ..., x_n]$. Empty lists are denoted by $[]$, the ith element x_i is also written $l[i]$ and the length of the list is $|l| = n$. An underscore is also used to indicate a list of values $\underline{x} = l$. Projection on lists are mapped to their elements, i.e. $\underline{x}^{\downarrow 1} = \left[x_1^{\downarrow 1}, ..., x_n^{\downarrow 1}\right]$.
- $\mathsf{sortOn} : i \to \mathcal{A}^* \to \mathcal{A}^*$ does sort a list of values on the ith projection.
- Data is a universal data type of nested sums and products built up recursively from the base types of $\mathbb{Z}$ and $\mathbb{H}$.

Note: The concatenation operator is defined in Agda directly in terms of bytes and concat (both law-free primitives; the definition fixes the encoding shape, it does not constrain the operands). Readers new to Agda may want Section 8.1, which maps the idioms used from here on to their Haskell counterparts.

3.2 Public key multi-signature scheme

A multisignature scheme is a set of algorithms where

- $\mathsf{MS\text{-}Setup}$ generates public parameters Π, such that
- $(k^{\text{ver}}, k^{\text{sig}}) \leftarrow \mathsf{MS\text{-}KG}(\Pi)$ can be used to generate fresh key pairs,

- $\sigma \leftarrow \mathsf{MS\text{-}Sign}(\Pi, k^{\text{sig}}, m)$ signs a message m using key k^{sig},
- $\tilde{k} \leftarrow \mathsf{MS\text{-}AVK}(\Pi, \overline{k})$ aggregates a list of verification keys $\overline{k}$ into a single, aggregate key $\tilde{k}$,
- $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(\Pi, m, \overline{k}, \overline{\sigma})$ aggregates a list of signatures $\overline{\sigma}$ about message m into a single, aggregate signature $\tilde{\sigma}$.
- $\mathsf{MS\text{-}Verify}(\Pi, \tilde{k}, m, \tilde{\sigma}) \in \mathbb{B}$ verifies an aggregate signature $\tilde{\sigma}$ of message m under an aggregate verification key $\tilde{k}$.

The security definition of a multisignature scheme from [5, 6] guarantees that, if $\tilde{k}$ is produced from a tuple of verification keys $\overline{k}$ via MS-AVK, then no aggregate signature $\tilde{\sigma}$ can pass verification $\mathsf{MS\text{-}Verify}(\tilde{k}, m, \tilde{\sigma})$ unless all honest parties holding keys in $\overline{k}$ signed m.

Note that in the following, we make the parameter Π implicit and leave out the *ver* suffix for verification key such that $k = k^{\text{ver}}$ for better readability.

In the Agda formalisation the scheme is instantiated directly at the protocol's types rather than packaged as a record: the verifier `msVfy` is an abstract function of an (aggregate) key, message and aggregate signature, used both by the on-chain signature conditions of Section 5 (`snapshotSigOK`) and by the security model of Section 7, where the security definition above is captured by the two assumptions `aggSound` (a verifying aggregate decomposes into verifying components) and `sigUnforge` (per-signature EUF-CMA), from which "a verifying aggregate means every party signed" (`ms-unforgeable`) is a derived theorem (Section 8).

3.3 Extended UTxO

The Hydra Head protocol is specified to work on the so-called Extended UTxO (EUTxO) ledgers like Cardano.

The basis for EUTxO is Bitcoin's UTxO ledger model [7, 8]. Intuitively, it arranges transactions in a directed acyclic graph, such as the one in Figure 2, where boxes represent transactions with (red) inputs to the left and (black) outputs to the right. A dangling (unconnected) output is an *unspent transaction output (UTxO)* — there are two UTxOs in the figure.

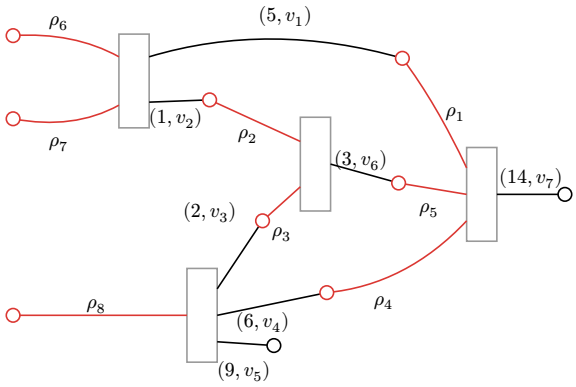

Figure 2: Example of a plain UTxO graph

The following paragraphs will give definitions of the UTxO model and it's extension to support scripting (EUTxO) suitable for this Hydra Head protocol specification. For a more detailed introduction to the EUTxO ledger model, see [3], [4] and [9].

3.3.1 Values

Definition 1 (Values). Values are sets that keep track of how many units of which tokens of which currency are available. Given a finitely supported function $\mapsto$, that maps keys to monoids, a value is the set of such mappings over currencies (minting policy identifiers), over a mapping of token names t to quantities q:

$$\mathsf{val} \in \mathsf{Val} = (c : \mathbb{H} \mapsto (t : \mathbb{H} \mapsto q : \mathbb{Z}))$$

where addition of values is defined as $+$ and $\varnothing$ is the empty value.

For example, the value $\{c_1 \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\}$ contains tokens t_1 and t_2 of currency c_1 and addition merges currencies and token names naturally:

$$\begin{aligned} & \{c_1 \mapsto \{t_1 \mapsto 1, t_2 \mapsto 1\}\} \\ + \ & \{c_1 \mapsto \{t_2 \mapsto 1, t_3 \mapsto 1\}, c_2 \mapsto \{t_1 \mapsto 2\}\} \\ = \ & \{c_1 \mapsto \{t_1 \mapsto 1, t_2 \mapsto 2, t_3 \mapsto 1\}, c_2 \mapsto \{t_1 \mapsto 2\}\} \ . \end{aligned}$$

While the above definition should be sufficient for the purpose of this specification, a full definition for finitely supported functions and values as used here can be found in [9]. To further improve readability, we define the following shorthands:

- $\{t_1, ..., t_n\} :: c$ for a set positive single quantity assets $\{c \mapsto \{t_1 \mapsto 1, ..., t_n \mapsto 1\}\}$,
- $\{t_1, ..., t_n\}^{-1} :: c$ for a set of negative single quantity assets $\{c \mapsto \{t_1 \mapsto -1, ..., t_n \mapsto -1\}\}$,
- $\{c \mapsto t \mapsto q\}$ for the value entry $\{c \mapsto \{t \mapsto q\}\}$,
- $\{c \mapsto \cdot \mapsto q\}$ for any asset with currency c and quantity q irrespective of token name.

3.3.2 Scripts

Validator scripts are called *phase-2* scripts in the Cardano Ledger specification (see [10] for a formal treatment of these). Scripts are used for multiple purposes, but most often (and sufficient for this specification) as a *spending* or *minting* policy script.

Definition 2 (Minting Policy Script). A script $\mu \in \mathcal{M}$ governing whether a value can be minted (or burned), is a pure function with type

$$\mu \in \mathcal{M} = (\rho : \mathsf{Data}) \to (\gamma : \Gamma) \to \mathbb{B},$$

where $\rho \in \mathsf{Data}$ is the redeemer provided as part of the transaction being validated and $\gamma \in \Gamma$ is the validation context.

Definition 3 (Spending Validator Script). A validator script $\nu \in \mathcal{V}$ governing whether an output can be spent, is a pure function with type

$$\nu \in \mathcal{V} = (\delta : \mathsf{Data}) \to (\rho : \mathsf{Data}) \to (\gamma : \Gamma) \to \mathbb{B},$$

where $\delta \in \mathsf{Data}$ is the datum available at the output to be spent, $\rho \in \mathsf{Data}$ is the redeemer data provided as part of the transaction being validated, and $\gamma \in \Gamma$ is the validation context.

3.3.3 Transactions

TODO: actual transactions $\mathcal{T}$ are not defined

We define EUTxO inputs, outputs and transactions as they are available to scripts and just enough to specify the behavior of the Hydra validator scripts. For example outputs addresses and datums are much more complicated in the full ledger model [4, 11].

Definition 4 (Outputs). An output $o \in \mathcal{O}$ stores some value $\mathsf{val} \in \mathsf{Val}$ at some address, defined by the hash of a validator script $\nu^{\#} \in \mathbb{H} = \mathsf{hash}(\nu \in \mathcal{V})$, and may store (reference) some data $\delta \in \mathsf{Data}$:

$$o \in \mathcal{O} = \left(\mathsf{val} : \mathsf{Val} \times \nu^{\#} : \mathbb{H} \times \delta : \mathsf{Data}\right)$$

Definition 5 (Output references). An output reference $\varphi \in \Phi$ points to an output of a transaction, using a transaction id (that is, a hash of the transaction body) and the output index within that transaction.

$$\varphi \in \Phi = (\mathbb{H} \times \mathbb{N})$$

Definition 6 (Inputs). A transaction input $i \in \mathcal{I}$ is an output reference $\varphi \in \Phi$ together with the *resolved* spent output (its value, address and datum, as in the eUTxO ledger) and a corresponding redeemer $\rho \in \mathsf{Data}$:

$$i \in \mathcal{I} = (\varphi : \Phi \times \mathsf{resolved} : \mathsf{Output} \times \rho : \mathsf{Data})$$

Definition 7 (Validation Context). A validation context $\gamma \in \Gamma$ is a view on the transaction to be validated:

$$\gamma \in \Gamma = (\mathcal{I}^* \times \mathcal{O}^* \times \mathsf{Val} \times \mathcal{S}^{\leftrightarrow} \times \mathcal{K})$$

where $\mathcal{I} \in \mathcal{I}^*$ is a (ledger-level) set of *resolved* inputs - each carries the spent output (its value, address and datum), as in the eUTxO ledger - modelled in Agda as a **list** so the value locked at a given script can be summed; $\mathcal{O} \in \mathcal{O}^*$ is a **list** of outputs, $\mathsf{mint} \in \mathsf{Val}$ is the minted (or burned) value, $(t_{\mathsf{min}}, t_{\mathsf{max}}) \in \mathcal{S}^{\leftrightarrow}$ are the lower and upper validity bounds where $t_{\mathsf{min}} \leq t_{\mathsf{max}}$, and $\kappa \in \mathcal{K}$ is the set of verification keys which signed the transaction. The context additionally exposes the hash of the validator script currently being run ($\mathsf{ownHash}$), against which a state-machine validator identifies its own continuing input and output (and hence the head value in/out).

As a first step of the machine-checked formalisation, these definitions are captured directly in Agda, with the value, script and key types kept abstract for now.

Informally, scripts are evaluated by the ledger when it applies a transaction to its current state to yield a new ledger state (besides checking the transaction integrity, signatures and ledger rules). Each validator script referenced by an output is passed its arguments drawn from the output it locks and the transaction context it is executed in. The transaction is valid if and only if all scripts validate, i.e. $\mu(\rho, \gamma) = \texttt{true}$ and $\nu(\delta, \rho, \gamma) = \texttt{true}$.

3.3.4 State machines and graphical notation

State machines in the EUTxO ledger model are commonly described using the *constraint emitting machine (CEM)* formalism [3], e.g. the original paper describes the Hydra Head protocol using this notation [1]. Although inspired by CEMs, this specification uses a more direct representation of individual transactions to simplify description of non-state-machine transactions and help translation to concrete implementations on Cardano. The structure of the state machine is

enforced on-chain through *scripts* which run as part of the ledger's validation of a transaction (see Section 3.3). For each protocol transaction, the specification defines the structure of the transaction and enumerates the transaction constraints enforced by the scripts ($\text{tx}^{\equiv}$ in the CEM formalism).

TODO: Add an example graph with a legend

3.4 KZG Accumulators

An accumulator scheme over a universe $\mathcal{U}$ is a set of algorithms where

- $\mathsf{accSetup}$ generates public parameters (the common reference string),
- $\eta \leftarrow \mathsf{accUTxO}(U)$ constructs an accumulator commitment from a UTxO set $U \subseteq \mathcal{U}$,
- $\eta' \leftarrow \mathsf{accCombine}(\eta_1, \eta_2)$ combines two accumulator commitments into one,
- $\pi \leftarrow \mathsf{accWitness}(\eta, S, U \setminus S)$ produces a membership witness π proving that $S \subseteq U$,
- $\mathsf{accVerify}(\eta, S, \pi) \in \mathbb{B}$ verifies a membership witness for subset S,
- $\eta' \leftarrow \mathsf{accExclude}(\eta, S)$ removes subset S from the accumulator, yielding the updated commitment η', which itself serves as the exclusion witness, and
- $\mathsf{accVerifyExclude}(\eta, S, \eta') \in \mathbb{B}$ verifies that η' is the correct accumulator after removing S from η.

The security property guarantees that a valid membership witness can only be produced for sets genuinely committed under η, and a valid exclusion witness can only be produced when the subset was genuinely removed.

In the Agda formalisation the scheme is likewise instantiated directly at the protocol's types (Section 5): `accUTxO`, `accVerify` and `accVerifyExclude` are abstract functions over UTxO-output sets, commitments and witnesses, and the security property above is captured by the specifying laws `accVerify-sound` (a verified membership witness attests a genuine subset) and `accVerify-self` (a set always verifies against its own commitment), which the fan-out safety and coverage theorems consume.

The implementation uses KZG polynomial commitments [12] over the BLS12-381 elliptic curve.

4 Protocol Setup

In order to create a head-protocol instance, an initiator invites a set of participants (the initiator being one of them) to join by announcing to them the protocol parameters.

- For on-chain transaction authentication (Cardano) purposes, each party p_i generates a corresponding key pair $\left(k_i^{\text{ver}}, k_i^{\text{sig}}\right)$ and sends their verification key k_i^{ver} to all other parties. In the case of Cardano, these are Ed25519 keys.
- For off-chain signing (Hydra) purposes, a very basic multisignature scheme (MS, as defined in Section 3.2) based on EdDSA using Ed25519 keys is used:
  - MS-KG is Ed25519 key generation (requires no parameters)
  - MS-Sign creates an EdDSA signature
  - MS-AVK is concatenation of verification keys into an ordered list
  - MS-ASig is concatenation of signatures into an ordered list
  - MS-Verify verifies the "aggregate" signature by verifying each individual EdDSA signature under the corresponding Ed25519 verification key

  To help distinguish on- and off-chain key sets, Cardano verification keys are written k_{C}, while Hydra verification keys are indicated as k_{H} for the remainder of this document.
- Each party p_i generates a hydra key pair and sends their hydra verification key to all other parties.
- Each party p_i computes the aggregate key from the received verification keys, stores the aggregate key, their signing key as well as the number of participants n.
- Each party establishes pairwise communication channels to all other parties. That is, every network message received from a specific party is checked for (channel) authentication. It is the implementer's duty to find a suitable authentication process for the communication channels.
- All parties agree on a contestation period T_{contest}.

If any of the above fails (or the party does not agree to join the head in the first place), the party aborts the initiation protocol and ignores any further action. Finally, at least one of the participants posts the *init* transaction onchain as described next in Section 5.

The parameters each party stores after a successful setup are captured by an Agda record (verification keys and the contestation period are kept abstract, the latter as a duration in time units).

5 On-chain Protocol

The following sections describe the *on-chain* protocol controlling the life-cycle of a Hydra head, which can be intuitively described as a state machine (see Figure 1). Each transition in this state machine is represented and caused by a corresponding Hydra protocol transaction on-chain: *init* Section 5.1, *increment* Section 5.4, *decrement* Section 5.5, *close* Section 5.6, *contest* Section 5.7, *fanout* Section 5.8, *partialFanout* Section 5.8.1, and *finalPartialFanout* Section 5.8.2.

The protocol uses KZG accumulators (see Section 3.4) to enable partial fanout when UTxO sets exceed transaction size limits. When all UTxOs fit in a single transaction, *fanout* distributes them all at once. When UTxO sets are too large, *partialFanout* distributes subsets across multiple transactions using membership witnesses, transitioning through an intermediate fanoutProgress state, until *finalPartialFanout* completes the distribution.

Besides the main state transitions of the head protocol, there is the related "deposit protocol" with two transactions in support of *increment*: *deposit* Section 5.2 and *recover* Section 5.3. There is also a *decrement* transaction Section 5.5 that allows for taking funds from the Head back to L1.

The head protocol defines one minting policy script and one validator script:

- μ_{head} governs minting of state and participation tokens in *init* and burning of these tokens in *fanout* or *finalPartialFanout*.
- ν_{head} represents the main protocol state machine logic and ensures contract continuity throughout *increment*, *decrement*, *close*, *contest*, *fanout*, *partialFanout* and *finalPartialFanout*.

The deposit protocol defines one validator script:

- ν_{deposit} controls that *deposit* transaction output is claimed correctly into a head via *increment* or recovered after the deadline has passed in a *recover* transaction.

The head output datum δ_{head} ranges over the protocol states. The state machine and its per-state fields (as enumerated in the transitions below) are captured by an Agda type, with the redeemer ρ_{head} selecting the ν_{head} transition.

The admissible ν_{head} state transitions are captured as a typed relation $d \longrightarrow \langle r \rangle d'$ ("datum d steps to d' under redeemer r"). The relation encodes the **state-machine shape** and the **version discipline** in the types: increment/decrement bump the version (`suc v`), close/contest preserve it (the same `v` reappears), close initialises the contester list to the empty list, contest requires the new $k^{\#} \notin \mathcal{C}$ (so the list grows by exactly one), the partial-fanout rules thread the intermediate fanoutProgress state through to final, and every rule reuses the same `ada` binder on both sides, so the ADA overhead $\mathbb{A}_o$ of Section 5.4–Section 5.7 is preserved by construction (there is no separate $\mathbb{A}_o$ conjunct in the bundles below). A rule violating any of **these** would fail to type-check. The remaining per-transaction conditions (signatures, value conservation, deadlines) are separate predicates (e.g. `closeDeadlineOK`/`contestDeadlineOK`) applied alongside it.

Beyond the state-machine shape, individual ν_{head} **checks** are stated as predicates over the validation Context and the datums. For example, the close transaction (Section 5.6) requires the recorded contestation deadline to be the transaction's upper validity bound extended by the contestation

period. This condition is stated as a checkable proposition over the context and the produced datum.

The shared trust surface - the accumulator scheme and its specifying laws, the value projections, the signature check - and the predicates common to several transactions are stated once here; each transaction's own conditions are then given in its section below.

Scope of the validity bundles. The per-transaction conditions of the following sections are *validity bundles*: records conjoining the state-machine step with the checks expressible from the datums, redeemer and context, inhabited exactly for genuinely valid transactions. What the bundles type-enforce is the state-machine shape, the version discipline, contester growth and deduplication, the deadline equations, close-initialises-to-$\emptyset$, the head value in/out (derived: `headValue`/`headValueIn` sum the value at `ownHash` over the produced outputs / resolved inputs), the increment deposit value (derived: `depositsValue` sums the value at the ν_{deposit} script `depHash` over *all* spent inputs, where Plutus `mustPreserveValue` uses the claimed deposit alone and rules the others out separately with `mustNotSpendOtherScripts`), the decrement decommit value (derived: `decommitValue` sums the `m` outputs after the head output, as Plutus `take m (tail outputs)`), and the participant signature: close, increment and decrement carry the structural `signedByParticipant cid ctx` (an existence witness `∃ kh` naming both a transaction signer and a participation token of the head value), while contest carries the sharper `contesterSigned`/`contesterIsParticipant` pair about the appended contester, from which `signedByParticipant` is derived (`contest-participantSigned`); fanout and partial fanout have no such field. What remains abstracted: the value arithmetic laws (`_+ᵛ_`/`εᵛ`) and the per-asset projection `quantityOf` on the opaque `Value`, crypto (`msVfy`/`snapshotSigOK`) and the accumulator operations (`accVerify`/`accVerifyExclude`/`accUTxO`), all via postulated laws, plus the context lookups the `Context` model does not expose: the hash of the commit list decoded from the claimed deposit's datum (`depositDatumCommitsHash`; the full commit digest `depositCommitsHashOf` is a definition over it, binding the deposit's transaction id) and the CRS reference-input datum hash with its canonical constant (`crsDatumHashAt`/`canonicalCRS#`). So value conservation is stated over the real head, increment-deposit and decrement-decommit values (modulo the abstract value algebra), while signature and accumulator soundness are assumed. None of this on-chain layer is rendered in Section 8: the datum/redeemer types, the transition relation, the postulated trust base, the helper predicates and the bundles themselves are all typechecked as part of this document's build but not shown — the appendix is reserved for the machine-checked theorem statements (Section 5.9, Section 6.5, Section 7.2), and the trust base is inventoried in prose in Section 8.2.

5.1 Init transaction

The *init* transaction creates a head instance and establishes the initial state of the protocol and is shown in Figure 3. The head instance is represented by the unique currency identifier cid created by minting tokens using the μ_{head} minting policy script which is parameterized by a single output reference parameter $\varphi_{\mathsf{seed}} \in \Phi$:

$$\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\varphi_{\mathsf{seed}}))$$

Two kinds of tokens are minted:

- A single *State Thread (ST)* token marking the head output. This output contains the state of the protocol on-chain and the token ensures contract continuity. The token name is the well known string `HydraHeadV2`, i.e. $\mathsf{ST} = \{\mathsf{cid} \mapsto \texttt{HydraHeadV2} \mapsto 1\}$.
- One *Participation Token (PT)* per participant $i \in \{1... \, |\underline{k}_\mathsf{H}|\}$ to be used for authenticating further transactions. The token name is the participant's verification key hash $k_i^\# = \mathsf{hash}(k_i^{\text{ver}})$ of the verification key as received during protocol setup, i.e. $\mathsf{PT}_i = \left\{\mathsf{cid} \mapsto k_i^\# \mapsto 1\right\}$.

All minted tokens (ST and all PT_i) are placed directly into the head output, which is created in the `open` state with an empty UTxO set. Consequently, the *init* transaction

- has at least input φ_seed,
- mints the state thread token ST, and one PT for each of the $|\underline{k}_\mathsf{H}|$ participants with policy cid,
- has one head output o_head, which captures the open state of the protocol in the datum, i.e. the `Open` constructor of `HeadDatum` (defined above) instantiated with
  - open is the state identifier,
  - cid' is the unique currency id of this instance,
  - $\underline{k}_\mathsf{H}$ are the off-chain multi-signature keys from the setup phase,
  - n is the number of participants,
  - T_contest is the contestation period,
  - $v = 0$ is the initial snapshot version,
  - $\eta = \mathsf{accUTxO}(\emptyset)$ is the accumulator commitment to the (empty) initial UTxO set (its hash $\eta^\# = \mathsf{hash}(\eta)$ is what later snapshot signatures attest to),
  - ₳_o is the ADA overhead: the head-output lovelace that belongs to no layer-two UTxO (the min-UTxO deposit), fixed here and preserved by every subsequent transition (Section 5.4–Section 5.7).

  The on-chain datum additionally carries the φ_seed output reference the policy is parameterised by, which μ_head's datum check below binds ($\varphi_\mathsf{seed} = \varphi'_\mathsf{seed}$) so that $\mathsf{cid} = \mathsf{hash}(\mu_\mathsf{head}(\varphi_\mathsf{seed}))$ can be re-derived from the datum alone. The Agda `Open` datum has no seed field, so that binding is one of the hand-reviewed items listed at the end of this section.

Implementation note (datum representation). The Agda `Open` datum carries the accumulator *commitment* η, whereas the on-chain implementation stores only its *hash* $\eta^\#$ in the open state (the `OpenDatum.accumulatorHash` field); the full commitment is materialised only in the closed / $\mathsf{fanoutProgress}$ states, where fan-out membership proofs require it. The two coincide for every open-state check, since those reference η only through $\mathsf{hash}(\eta)$ (e.g. the close / increment / decrement signature over $\mathsf{cid} \parallel v \parallel s \parallel \eta^\# \parallel \delta^\# \parallel \kappa^\#$). Similarly, the datum's `hydraKey` field is the single *aggregate* key of the multisignature scheme rather than the per-party key list $\underline{k}_\mathsf{H}$; the checks written $\mathsf{MS\text{-}Verify}(\underline{k}_\mathsf{H}, ...)$ in Section 5.4–Section 5.7 are verified under this aggregate key (Agda `hk`).

The $\mu_\mathsf{head}(\varphi_\mathsf{seed})$ minting policy is the only script that verifies init transactions and can be redeemed with either a Mint or Burn redeemer:

- When evaluated with the Mint redeemer,

1. The seed output is spent in this transaction. This guarantees uniqueness of the policy cid because the EUTxO ledger ensures that φ_{seed} cannot be spent twice in the same chain. $(\varphi_{\mathsf{seed}}, \cdot) \in \mathcal{I}$
2. All minted tokens of this policy are of single quantity $\forall\{\mathsf{cid} \mapsto \cdot \mapsto q\} \in \mathsf{mint} : q = 1$. (The policy counts only its own currency and enforces unit quantities indirectly, via the head-output checks below; mint entries of *other* policies are not constrained by μ_{head} - they validate themselves.)
3. Right number of tokens of *this* policy are minted $|\{\mathsf{cid} \mapsto \cdot \mapsto q\} \in \mathsf{mint}| = |\underline{k}_{\mathsf{H}}| + 1$ (one ST plus one PT per member; as above, entries of other policies are outside μ_{head}'s count)
4. State token is sent to the head validator $\mathsf{ST} \in \mathsf{val}_{\mathsf{head}}$
5. All participation tokens are sent to the head output alongside the state token $\forall i \in [1... \, |\underline{k}_{\mathsf{H}}|] : \mathsf{PT}_i \in \mathsf{val}_{\mathsf{head}}$
6. The δ_{head} contains own currency id $\mathsf{cid} = \mathsf{cid}'$ and the right seed reference $\varphi_{\mathsf{seed}} = \varphi'_{\mathsf{seed}}$

- When evaluated with the Burn redeemer,
  1. All tokens for this policy in mint need to be of negative quantity $\forall\{\mathsf{cid} \mapsto \cdot \mapsto q\} \in \mathsf{mint} : q < 0$. (Formalised as the `BurnValid` bundle below; *which* burns are legitimate is ν_{head}'s concern, via the fan-out family's $n + 1$ burn count.)

Important: The μ_{head} minting policy only ensures uniqueness of cid and that the right amount of tokens have been minted and sent to ν_{head}, while ν_{head} in turn ensures continuity of the contract. However, it is **crucial** that all head members check:

- That the transaction mints exactly the correct tokens: one ST token and one PT for each head member (total $|\underline{k}_{\mathsf{H}}| + 1$ tokens). This distinguishes *init* from *increment* and *decrement* transactions, which only move tokens without minting.
- That the head output contains an ST token of policy cid which satisfies $\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\varphi_{\mathsf{seed}}))$. The φ_{seed} spent by this transaction can be used to determine this.
- That the correct verification key hashes are used in the PTs and the open state is consistent with parameters agreed during setup.

See the initialTx behavior in Figure 13 for details about these checks. The decidable core of these checks is formalised as the `initValid` predicate (a *creation* predicate - init has no predecessor datum, so it is not a $\_\rightarrow\langle\_\rangle\_$ step): $\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\varphi_{\mathsf{seed}}))$, the seed is spent (a structural conjunct, `seedSpent`), exactly $n + 1$ tokens of cid are minted, and the produced Open is initial ($v = 0$, $\eta = \mathsf{accUTxO}(\emptyset)$). Token placement into the head value is also modelled (the state token is present and the head output carries exactly the $n + 1$ head-policy tokens). What remains hand-reviewed: the datum bindings - $\mathsf{cid} = \mathsf{hash}(\mu_{\mathsf{head}}(\varphi_{\mathsf{seed}}))$ is stated over the law-free `hash`, and the datum's seed-reference field $\varphi_{\mathsf{seed}} = \varphi'_{\mathsf{seed}}$ has no Agda counterpart (the `Open` datum carries no seed field).

The conditions are conjoined in the `InitValid` bundle with its dispatching `initValid` predicate (typechecked, not rendered).

The Burn arm's single check is the `BurnValid` bundle: every head-policy entry of the mint field is a burn - no head-policy token is minted and at least one is burned (the policy only runs when its currency appears in the mint field, and rejects a mint field without head-policy entries). It is

bridged (`burnValid→ref`) and differentially tested (the `HeadValidatorAgreement` burn agreement, calling the real `validateTokensBurning`).

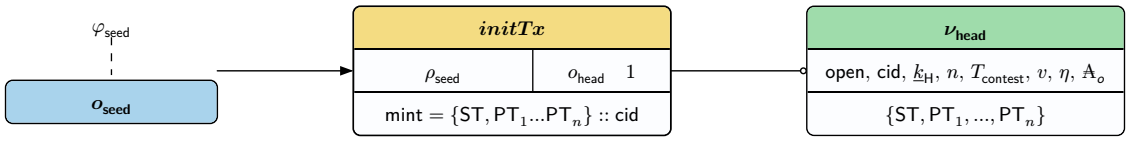

Figure 3: *init* transaction spending a seed UTxO and producing the head output directly in state open.

5.2 Deposit Transaction

The *deposit* transaction locks funds in ν_{deposit} for later collection into the head via an *increment* transaction. Any transaction paying to ν_{deposit} is a *deposit* transaction as there is no on-chain verification in *deposit* transactions. Consequently, protocol actors **must ensure off-chain** that a valid datum is used when paying to the ν_{deposit} validator. This is sufficient because a deposit is inert until claimed: on-chain validation at creation time is not even expressible (validators run on spend, not on receive), no head funds are exposed by a deposit's existence, and collection requires an *increment* transaction carrying the n-of-n multisignature (Section 5.4) — whose message binds the claimed deposit's exact commit set and the identity of the transaction that created it, recomputed on-chain from its datum and the claimed reference. Every honest node refuses to treat a deposit as claimable unless its datum decodes and its declared commits total exactly the value locked — a mismatch would otherwise make the head's accumulator insolvent and the head unfanoutable — so a single honest party blocks any malformed deposit: the same unanimity gate that authenticates all other head content. A deposit whose datum does not decode is unspendable by both the claim and recover arms (Section 5.3); only the depositor's own funds are at risk from a malformed datum.

A valid deposit output is governed by ν_{deposit} with value $\mathsf{val}_{\text{deposit}}$ and datum

$$\delta_{\text{deposit}} = (\mathsf{cid}, t_{\text{recover}}, C)$$

where

- cid is the currency id of the target head protocol instance (see Section 5.1),
- t_{recover} is a deadline after which the deposit can be recovered, and
- $C \in (\mathcal{J} \times \mathbb{H})^*$ is a list of transaction output references with along with serialized outputs that should become available in the head.

In Agda the deposit datum and the ν_{deposit} redeemer are the `DepositDatum` / `DepositRedeemer` types; the deposit transaction itself has no on-chain verification, so there is no corresponding validity bundle.

Head protocol participants determine **off-chain** whether a deposit output o_{deposit} is eligible for their head by checking

1. cid matches their head identifier,
2. t_{recover} is reasonably far in the future, and **TODO:** explain; relate to contestation period?
3. $\mathsf{val}_{\text{deposit}}$ contains the value of all decoded outputs of C from δ_{deposit}.

An example transaction which records all its spent inputs in a deposit output is shown in Figure 4. The $j \in \{1...m\}$ inputs of this example with reference $\varphi_{\mathsf{deposited}_j}$ each spend output $o_{\mathsf{deposited}_j}$ with $\mathsf{val}_{\mathsf{deposited}_j}$ would be recorded in the output datum as

$$C = \forall j \in \{1...m\} : \left[\left(\varphi_{\mathsf{deposited}_j}, \mathsf{bytes}\left(o_{\mathsf{deposited}_j}\right)\right)\right]$$

and the value check would need to satisfy

$$\mathsf{val}_{\mathsf{deposit}} \supseteq \bigcup_{j=1}^{m} \mathsf{val}_{\mathsf{deposited}_j}$$

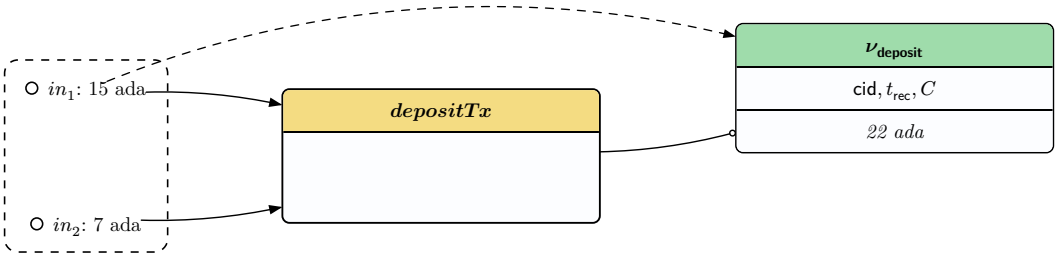

Figure 4: *deposit* transaction spending multiple UTxO into a deposit output.

5.3 Recover Transaction

If a *deposit* transaction output (see Section 5.2) was not collected into a head by an *increment* transaction Section 5.4, the *recover* transaction (Figure 5) allows for restoring the UTxO as recorded in the deposit after the deadline has passed. It consists of

- one input spending from ν_{deposit} with datum $\delta_{\mathsf{deposit}} = (\mathsf{cid}, t_{\mathsf{recover}}, C)$, and
- outputs $o_1...o_m$ to recover UTxOs.

The script validator ν_{deposit} is spent with redeemer $\rho_{\mathsf{deposit}} = (\mathsf{Recover}, m)$, where m is the number of outputs to recover, and checks:

1. All UTxOs are recovered exactly as they were deposited. This is done by comparing hashes of serialised representations of the m recovering outputs $o_1...o_m$ with the canonically combined deposited UTxOs in C:

$$\mathsf{hash}\left(\bigoplus_{j=1}^{m} \mathsf{bytes}(o_j)\right) = \mathsf{hash}\left(\mathsf{concat}\left(\mathsf{sortOn}(1, C)^{\downarrow 2}\right)\right)$$

2. Transaction is posted after the deadline

$$t_{\mathsf{min}} > t_{\mathsf{recover}}$$

3. This deposit is the only one spent by the transaction. The recovered outputs $o_1...o_m$ are the transaction's first m outputs and are therefore shared by every deposit input, so two deposits whose C hashes alike would both be satisfied by a single set of recovered outputs, leaving the second deposit's value to be directed anywhere by the transaction author. Note that ν_{deposit} neither parses the bytes of C nor relates them to the value the deposit holds, so hashing alike is weaker than being equal:

$$\left|\left\{(\cdot, o) \in \mathcal{I} \mid o \text{ is governed by } \nu_{\mathsf{deposit}}\right\}\right| = 1$$

The deposit datum and redeemer are formalised as `DepositDatum` / `DepositRedeemer`, and the recover checks as the `recoverValid` predicate: the recovered outputs match the deposited ones (`recoveredMatchesDeposited`, the Section 5.3 serialisation-hash equality, abstracted) and the transaction is posted strictly after the deadline (`t_recover < txValidityMin`, concrete). A deposit's collection into the head (Claim) is authorised by the `incrementValid` predicate's `depositSpentOK` check (Section 5.4); `depositClaimedBy` records that the deposit's `cid` must match the head it is claimed into.

The recover conditions form the `RecoverValid` bundle (typechecked, not rendered).

The Claim arm's own checks - the head binding and the before-deadline check, the two checks ν_{deposit} performs that ν_{head} does not - form the `ClaimValid` bundle, consumed by the joint claim obligation stated in Section 5.4.

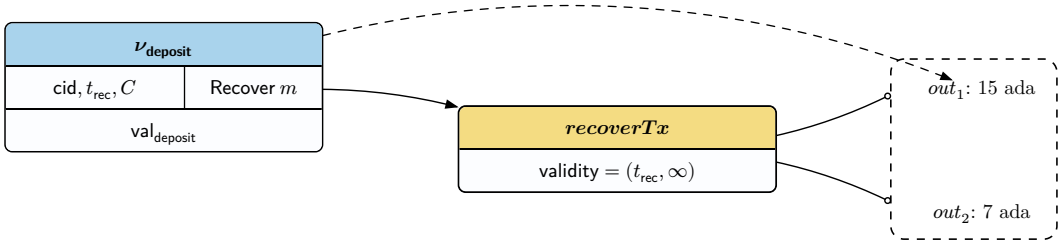

Figure 5: *recover* transaction restoring UTxO of a deposit output.

5.4 Increment Transaction

The *increment* transaction (Figure 6) allows a participant to add a *deposit* output Section 5.2 to an already open head using a snapshot that approves inclusion. Consequently this transaction consists of:

- one input spending from ν_{head} with value $\mathsf{val}_{\mathsf{head}}$ holding the ST and datum δ_{head},
- one input $\varphi_{\mathsf{deposit}}$ spending from ν_{deposit} with value $\mathsf{val}_{\mathsf{deposit}}$ and datum $\delta_{\mathsf{deposit}} = (\mathsf{cid}_{\mathsf{deposit}}, t_{\mathsf{recover}}, C)$,
- one output paying to ν_{head} with value $\mathsf{val}'_{\mathsf{head}}$ and datum δ'_{head}.

The deposit validator ν_{deposit} is spent with $\rho_{\mathsf{deposit}} = \mathsf{Claim}$ and ensures:

1. Claiming head id matches the deposit datum
$$\mathsf{cid} = \mathsf{cid}_{\mathsf{deposit}}$$
2. Transaction is posted before the deadline
$$t_{\mathsf{max}} \leq t_{\mathsf{recover}}$$
3. The head input is spent with an $\mathsf{increment}$ redeemer that claims this very deposit. Otherwise a legitimate *increment* could carry additional deposit inputs whose value enters the head without any snapshot crediting it:
$$\varphi_{\mathsf{increment}} = \varphi_{\mathsf{deposit}}$$

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{increment}, \xi, s, \varphi_{\mathsf{increment}}, \delta^{\#})$, where ξ is a multi-signature of the increment snapshot which authorizes addition of deposited UTxO, s is

the snapshot number, $\varphi_{\mathsf{deposit}}$ points to the claimed deposit and $\delta^{\#}$ is the hash of the snapshot's decommit output set. The validator checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{open}$ to $\delta'_{\mathsf{head}} \sim \mathsf{open}$, parameters $\mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}$ stay unchanged and the new state is governed again by ν_{head}:

$$(\mathsf{open}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}, v, \eta, \cancel{\mathbb{A}}_o) \xrightarrow[\xi, s, \varphi_{\mathsf{increment}}, \delta^{\#}]{\mathsf{increment}} (\mathsf{open}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}, v', \eta', \cancel{\mathbb{A}}_o)$$

   (the `increment` rule of `_→⟨_⟩_`; the version is bumped, $v \mapsto v + 1$).

2. New version v' is incremented correctly

$$v' = v + 1$$

3. Claimed deposit is spent

$$\varphi_{\mathsf{increment}} = \varphi_{\mathsf{deposit}}$$

4. The claimed deposit is the first output of its transaction, as produced by *deposit* (Section 5.2). Since $\kappa^{\#}$ below binds a deposit by its transaction id, sibling outputs of that same transaction would otherwise be interchangeable under one signature:

$$txIdx(\varphi_{\mathsf{deposit}}) = 0$$

5. ξ is a valid multi-signature of the new head state η'

$$\mathsf{MS\text{-}Verify}\big(\underline{k}_{\mathsf{H}}, \big(\mathsf{cid} \parallel v \parallel s \parallel (\eta')^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}\big), \xi\big) = \texttt{true}$$

   where $(\eta')^{\#} = \mathsf{hash}(\eta')$ is the hash of the new accumulator commitment η' stored in the output datum, reflecting the UTxO set after adding the deposited UTxOs; $\delta^{\#}$ is taken from the redeemer; and $\kappa^{\#}$ is *recomputed on-chain* from the claimed deposit itself, binding both the commit list C decoded from its own datum $\delta_{\mathsf{deposit}} = \big(\mathsf{cid}_{\mathsf{deposit}}, t_{\mathsf{recover}}, C\big)$ (a deposit datum that fails to decode rejects the transaction, error `DepositDatumInvalid`) and the identity of the transaction that created it:

$$\kappa^{\#} = \mathsf{hash}\big(\mathsf{hash}\big(\mathsf{concat}\big(\mathsf{sortOn}(1, C)^{\downarrow 2}\big)\big) \parallel txId(\varphi_{\mathsf{deposit}})\big)$$

   Hashing C alone would not identify a deposit: a deposit datum is unauthenticated data that anyone can copy into a look-alike deposit holding less value, which hashes identically and would accept the same signature.

6. The value in the head output is increased accordingly

$$\mathsf{val}_{\mathsf{head}} \oplus \mathsf{val}_{\mathsf{deposit}} = \mathsf{val}'_{\mathsf{head}}$$

7. Transaction is signed by a participant

$$\exists \big\{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\big\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$$

8. No minting or burning

$$\mathsf{mint} = \emptyset$$

   (the bundle's `mintEmpty` field)

9. The ADA overhead $\cancel{\mathbb{A}}_o$ is preserved across the state transition:

$$\mathbb{A}'_o = \mathbb{A}_o$$

(enforced by the `step` field's type: the `increment` rule reuses the same `ada` binder on both sides)

These conditions form the `IncrementValid` bundle, over the additive value-conservation predicate `incrementValueOK`.

A deposit claim must satisfy both validators run in the same transaction: ν_{head}'s `incrementValid` above and ν_{deposit}'s `claimValid` (Section 5.3). The joint claim obligation is stated here as `ClaimTxValid`, since it conjoins `incrementValid` with the deposit-side bundle.

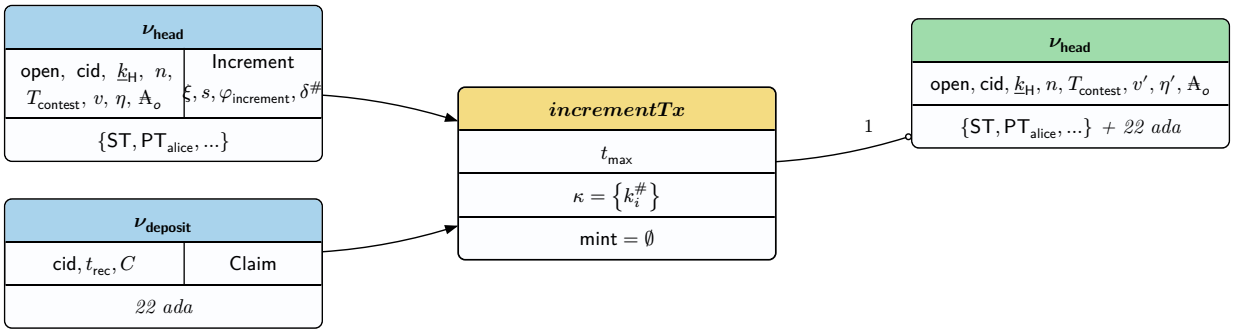

Figure 6: *increment* transaction spending an open head output, producing a new head output which includes the new UTxO.

5.5 Decrement Transaction

The *decrement* transaction (Figure 7) allows a party to remove a UTxO from an already open head and consists of:

- one input spending from ν_{head} holding the ST with δ_{head},
- one output paying to ν_{head} with value $\mathsf{val}'_{\mathsf{head}}$ and datum δ'_{head},
- one or more decommit outputs $o_2 \ldots o_{m+1}$ with values $\mathsf{val}_2 \ldots \mathsf{val}_{m+1}$.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{decrement}, \xi, s, m, \kappa^{\#})$, where ξ is a multi-signature of the decrement snapshot which authorizes removal of some UTxO, s is the used snapshot number, m is the number of outputs to distribute and $\kappa^{\#}$ is the hash of the snapshot's commit output set. The validator checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{open}$ to $\delta'_{\mathsf{head}} \sim \mathsf{open}$, parameters $\mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}$ stay unchanged and the new state is governed again by ν_{head}

   $$(\mathsf{open}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}, v, \eta, \mathbb{A}_o) \xrightarrow[\xi, s, m, \kappa^{\#}]{\mathsf{decrement}} (\mathsf{open}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}, v', \eta', \mathbb{A}_o)$$

   (the `decrement` rule of `_→⟨_⟩_`; the version is bumped, $v \mapsto v + 1$).
2. New version v' is incremented correctly

   $$v' = v + 1$$
3. At least one decommit output is materialized. Note that this is checked on the outputs actually present, not on the redeemer's m: the transaction's decommit outputs are taken as the first m after the head output, which truncates silently when fewer exist.

   $$|o_2 \ldots o_{m+1}| > 0$$

This is required because *increment* and *decrement* verify the same signed message. With no decommit outputs, $\delta^{\#}$ recomputed here is the digest of the empty set — exactly what a snapshot carrying no pending decommit produces — while $\kappa^{\#}$ comes from the redeemer. An *increment* snapshot's multi-signature would otherwise authorize a decrement that advances the version and adopts that snapshot's accumulator, which already counts the deposited UTxOs, while no deposit is spent and no value enters the head. The head would then account for UTxOs it does not hold.

4. ξ is a valid multi-signature of the new snapshot state η'

$$\mathsf{MS\text{-}Verify}\Big(\underline{k}_{\mathsf{H}}, \big(\mathsf{cid} \parallel v \parallel s \parallel (\eta')^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}\big), \xi\Big) = \texttt{true}$$

   where $(\eta')^{\#} = \mathsf{hash}(\eta')$ is the hash of the new accumulator commitment η' stored in the output datum, reflecting the UTxO set after removing the decommitted UTxOs; $\kappa^{\#}$ is taken from the redeemer; and $\delta^{\#}$ is *recomputed on-chain* as the hash of the m decommit outputs $o_2 ... o_{m+1}$ following the head output — the same output list the value check below sums. Binding $\delta^{\#}$ to the exact decommit output set (address, datum, order and count, not just aggregate value) means a signer cannot redirect decommitted outputs while reusing a valid signature.

5. The value in the head output is decreased accordingly

$$\mathsf{val}'_{\mathsf{head}} \oplus \left(\bigoplus_{j=2}^{m+1} \mathsf{val}_j \right) = \mathsf{val}_{\mathsf{head}}$$

6. Transaction is signed by a participant

$$\exists \Big\{ \mathsf{cid} \mapsto k_i^{\#} \mapsto 1 \Big\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$$

7. No minting or burning

$$\mathsf{mint} = \emptyset$$

   (the bundle's `mintEmpty` field)

8. The ADA overhead $\mathbb{A}_o$ is preserved across the state transition:

$$\mathbb{A}'_o = \mathbb{A}_o$$

   (enforced by the `step` field's type: the `decrement` rule reuses the same `ada` binder on both sides)

These conditions form the `DecrementValid` bundle.

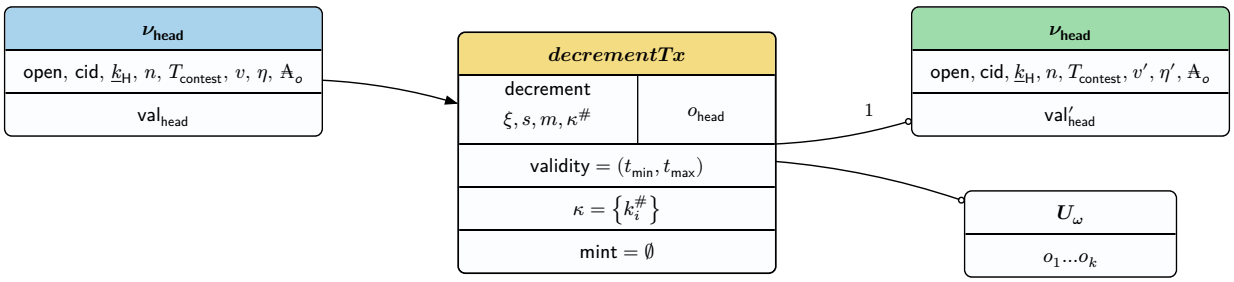

Figure 7: *decrement* transaction spending an open head output, producing a new head output and multiple decommitted outputs.

5.6 Close Transaction

In order to close a head, a head member may post the *close* transaction (see Figure 8). This transaction has

- one input spending from ν_{head} holding the ST with δ_{head},
- one output paying to ν_{head} with value $\text{val}'_{\text{head}}$ and datum δ'_{head}.

The state-machine validator ν_{head} is spent with $\rho_{\text{head}} = (\text{close}, \text{CloseType})$, where CloseType is a hint against which open state to close. (The closing party posts $\text{postTx}\big(\mathit{close}, \hat{v}, \bar{\mathcal{S}}.v, \bar{\mathcal{S}}.s, (\eta')^{\#}, \xi\big)$ off-chain; on-chain the redeemer carries only $\big(\xi, (\eta')^{\#}, \delta^{\#}, \kappa^{\#}\big)$ in CloseType — the signature, the accumulator hash and the decommit- and commit-output-set hashes — while the version v and snapshot number s are authenticated by the multisignature ξ over $\text{cid} \parallel v \parallel s \parallel (\eta')^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}$ and recorded in the datum, rather than being separate redeemer fields.) The transaction checks:

1. State is advanced from $\delta_{\text{head}} \sim \text{open}$ to $\delta'_{\text{head}} \sim \text{closed}$, parameters $\text{cid}, \underline{k}_{\text{H}}, n, T_{\text{contest}}$ stay unchanged and the new state is governed again by ν_{head}

   $$(\text{open}, \text{cid}, \underline{k}_{\text{H}}, n, T_{\text{contest}}, v, \eta, \mathbb{A}_o) \xrightarrow[\text{CloseType}]{\text{close}} (\text{closed}, \text{cid}, \underline{k}_{\text{H}}, n, T_{\text{contest}}, v', s', \eta', \mathcal{C}, t_{\text{final}}, \mathbb{A}_o)$$

   (the `close` rule of _→⟨_⟩_; the version is preserved, $v' = v$). The closed state carries a single unified accumulator η' that combines the snapshotted UTxO set with any pending increment or decrement UTxOs using accCombine.

2. Last known open state version is recorded in closed state

   $$v' = v$$

3. Based on the redeemer $\text{CloseType} = \text{Initial} \cup \big(\text{Any}, \xi, (\eta')^{\#}, \delta^{\#}, \kappa^{\#}\big) \cup \big(\text{Unused}, \xi, (\eta')^{\#}, \delta^{\#}, \kappa^{\#}\big) \cup \big(\text{Used}, \xi, (\eta')^{\#}, \delta^{\#}, \kappa^{\#}\big)$, where ξ is a multi-signature of the closing snapshot, $(\eta')^{\#}$ is the hash of the unified accumulator commitment stored in the output datum, and $\delta^{\#}/\kappa^{\#}$ are the snapshot's decommit-/commit-output-set hashes (passed verbatim into the signed message; they are authenticated only through ξ), four cases are distinguished. In each case the closed state carries a single unified accumulator η':

   1. Initial: The initial snapshot is used to close the head and open state was not updated. No signatures are available and it suffices to check

      $$v = 0$$

      $$s' = 0$$

$$\eta' = \mathsf{accUTxO}(\emptyset)$$

2. Any: Closing snapshot refers to current state version v with no pending increments or decrements, and $s' > 0$. The unified accumulator is simply the snapshotted state:

$$\eta' = \mathsf{accUTxO}(U')$$

$$\mathsf{MS\text{-}Verify}\Big(\underline{k}_{\mathsf{H}}, \big(\mathsf{cid} \parallel v \parallel s' \parallel (\eta')^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}\big), \xi\Big) = \texttt{true}$$

$$(\eta')^{\#} = \mathsf{hash}(\eta')$$

3. Unused: Closing snapshot refers to current state version v and a pending increment or decrement is *not* applied in the snapshot. The unified accumulator is the snapshotted state only:

$$\eta' = \mathsf{accUTxO}(U')$$

$$\mathsf{MS\text{-}Verify}\Big(\underline{k}_{\mathsf{H}}, \big(\mathsf{cid} \parallel v \parallel s' \parallel (\eta')^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}\big), \xi\Big) = \texttt{true}$$

$$(\eta')^{\#} = \mathsf{hash}(\eta')$$

4. Used: Closing snapshot refers to the previous state version $v - 1$ and a pending increment or decrement *is* applied in the snapshot. The unified accumulator combines the snapshotted UTxOs with the pending delta:

$$\eta' = \mathsf{accCombine}(\mathsf{accUTxO}(U'), \eta_{\Delta})$$

$$\mathsf{MS\text{-}Verify}\Big(\underline{k}_{\mathsf{H}}, \big(\mathsf{cid} \parallel v - 1 \parallel s' \parallel (\eta')^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}\big), \xi\Big) = \texttt{true}$$

$$(\eta')^{\#} = \mathsf{hash}(\eta')$$

where η_{Δ} is the accumulator commitment of the pending delta UTxOs.

4. Initializes the set of contesters

$$\mathcal{C} = \emptyset$$

This allows the closing party to also contest and is required for use cases where pre-signed, valid in the future, close transactions are used to delegate head closing.

5. Correct contestation deadline is set

$$t_{\mathsf{final}} = t_{\mathsf{max}} + T$$

6. Transaction validity range is bounded by

$$t_{\mathsf{max}} - t_{\mathsf{min}} \leq T$$

to ensure the contestation deadline t_{final} is at most $2 * T$ in the future.

7. Value in the head is preserved exactly

$$\mathsf{val}'_{\mathsf{head}} = \mathsf{val}_{\mathsf{head}}$$

8. Transaction is signed by a participant

$$\exists\left\{\mathsf{cid} \mapsto k_i^{\#} \mapsto 1\right\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$$

9. No minting or burning

$$\mathsf{mint} = \emptyset$$

10. The ADA overhead $\mathbb{A}_o$ is propagated unchanged from the open datum to the closed datum (enforced by the `close` rule's shared `ada` binder):

$$\mathbb{A}'_o = \mathbb{A}_o$$

where $\mathbb{A}_o$ is the ADA in the head UTxO not belonging to any L2 UTxO (minimum-UTxO overhead), set at initialisation time and unchanged for the head's lifetime. On fanout, the on-chain value conservation check treats $\mathbb{A}_o$ as released from the head UTxO without requiring it in any distributed output, so it flows to whoever submits the fanout transaction as change (offsetting their transaction fee).

Implementation note (accumulator construction). The per-case formulas for η' above ($\mathsf{accUTxO}(U')$ for Any/Unused, $\mathsf{accCombine}(\mathsf{accUTxO}(U'), \eta_\Delta)$ for Used) describe how the closing party constructs the unified accumulator *off-chain* before signing: ν_{head} has neither U' nor η_Δ and does not recompute them. On-chain it verifies only the multisignature ξ over $\mathsf{cid} \parallel v \parallel s' \parallel (\eta')^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}$ and the binding $(\eta')^{\#} = \mathsf{hash}(\eta')$; the Initial case additionally fixes $\eta' = \mathsf{accUTxO}(\emptyset) = G_1$ (a constant). The Agda `closeValid` bundle mirrors exactly this on-chain view - `closeSigOK` (the multisignature, at v or $v-1$ for Used), `closeηOK` ($(\eta')^{\#} = \mathsf{hash}(\eta')$), and `closeInitialOK` ($\eta = \mathsf{accUTxO}(\emptyset)$) - and likewise does not recompute the off-chain accUTxO/accCombine constructions, which are authenticated by ξ.

The close checks are formalised per condition - the deadline equation, the Initial-case constraint, the η-hash binding, the positive-snapshot Any case and the version-dependent signature obligation - and conjoined in the `CloseValid` bundle.

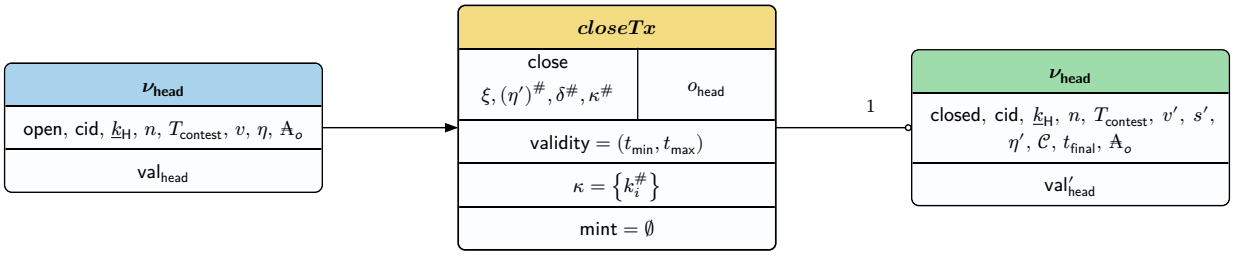

Figure 8: *close* transaction spending the open head output and producing a closed head output with unified accumulator η'.

5.7 Contest Transaction

The *contest* transaction (see Figure 9) is posted by a party to prove the currently closed state is not the latest one. This transaction has

- one input spending from ν_{head} holding the ST with δ_{head},
- one output paying to ν_{head} with value $\mathsf{val}'_{\mathsf{head}}$ and datum δ'_{head}.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{contest}, \mathsf{ContestType})$, where ContestType is a hint how to verify the snapshot and checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{closed}$ to $\delta'_{\mathsf{head}} \sim \mathsf{closed}$, parameters $\mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}$ stay unchanged and the new state is governed again by ν_{head}

$$(\mathsf{closed}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}, v, s, \eta, \mathcal{C}, t_{\mathsf{final}}, \mathbb{A}_o) \xrightarrow[\mathsf{ContestType}]{\mathtt{contest}} (\mathsf{closed}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}, v', s', \eta', \mathcal{C}', t'_{\mathsf{final}}, \mathbb{A}_o)$$

   (the `contest` rule of `_→⟨_⟩_`; the version is preserved and the contester set grows by one key, $\mathcal{C}' = \mathcal{C} \cup \{k^{\#}\}$). The closed state carries a single unified accumulator η' computed using $\mathsf{accCombine}$ based on the contest type.
2. Last known open state version stays recorded in closed state

$$v' = v$$

3. Contested snapshot number s' is higher than the currently stored snapshot number s

$$s' > s$$

4. Based on the redeemer $\mathsf{ContestType} = \left(\mathsf{Unused}, \xi, (\eta')^{\#}, \delta^{\#}, \kappa^{\#}\right) \cup \left(\mathsf{Used}, \xi, (\eta')^{\#}, \delta^{\#}, \kappa^{\#}\right)$, where ξ is a multi-signature of the contesting snapshot, $(\eta')^{\#} = \mathsf{hash}(\eta')$ is the hash of the unified accumulator commitment stored in the output datum, and $\delta^{\#}/\kappa^{\#}$ are the snapshot's decommit-/commit-output-set hashes (passed verbatim into the signed message, as for close), two cases are distinguished:

   1. Unused: Contesting snapshot refers to current state version v (pending delta not applied in snapshot). The unified accumulator reflects only the snapshotted UTxOs:

$$\eta' = \mathsf{accUTxO}(U')$$

$$\mathsf{MS\text{-}Verify}\left(\underline{k}_{\mathsf{H}}, \left(\mathsf{cid} \parallel v \parallel s' \parallel (\eta')^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}\right), \xi\right) = \mathtt{true}$$

$$(\eta')^{\#} = \mathsf{hash}(\eta')$$

   2. Used: Contesting snapshot refers to the previous state version $v - 1$ (pending delta applied in snapshot). The unified accumulator combines the snapshotted UTxOs with the pending delta:

$$\eta' = \mathsf{accCombine}(\mathsf{accUTxO}(U'), \eta_{\Delta})$$

$$\mathsf{MS\text{-}Verify}\left(\underline{k}_{\mathsf{H}}, \left(\mathsf{cid} \parallel v - 1 \parallel s' \parallel (\eta')^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}\right), \xi\right) = \mathtt{true}$$

$$(\eta')^{\#} = \mathsf{hash}(\eta')$$

      where η_{Δ} is the accumulator commitment of the pending delta UTxOs.
5. The single signer $\{k^{\#}\} = \kappa$ has not already contested and is added to the set of contesters

$$k^{\#} \notin \mathcal{C}$$

$$\mathcal{C}' = \mathcal{C} \cup k^{\#}$$

6. Transaction is posted before deadline

$$t_{\mathsf{max}} \leq t_{\mathsf{final}}$$

7. Contestation deadline is updated correctly to

$$t'_{\mathsf{final}} = \begin{cases} t_{\mathsf{final}} & \text{if} \sim |\mathcal{C}'| = n, \\ t_{\mathsf{final}} + T & \text{otherwise.} \end{cases}$$

8. Value in the head is preserved exactly

$$\mathsf{val}'_{\mathsf{head}} = \mathsf{val}_{\mathsf{head}}$$

9. Transaction is signed by a participant

$$\exists \left\{ \mathsf{cid} \mapsto k_i^{\#} \mapsto 1 \right\} \in \mathsf{val}'_{\mathsf{head}} \Rightarrow k_i^{\#} \in \kappa$$

10. No minting or burning

$$\mathsf{mint} = \emptyset$$

11. The ADA overhead $\mathbb{A}_o$ is preserved in the output closed datum (enforced by the `contest` rule's shared `ada` binder):

$$\mathbb{A}'_o = \mathbb{A}_o$$

The contest checks - the conditional deadline update, the η-hash binding and the version-dependent signature obligation - form the `ContestValid` bundle; the shared signed-by-a-participant obligation is derived from its sharper contester fields. NB the bundle's `contesterSigned` captures that the appended contester is *a* transaction signer; the implementation's stronger sole-signer cardinality ($\{k^{\#}\} = \kappa$, exactly one signer) is not modelled.

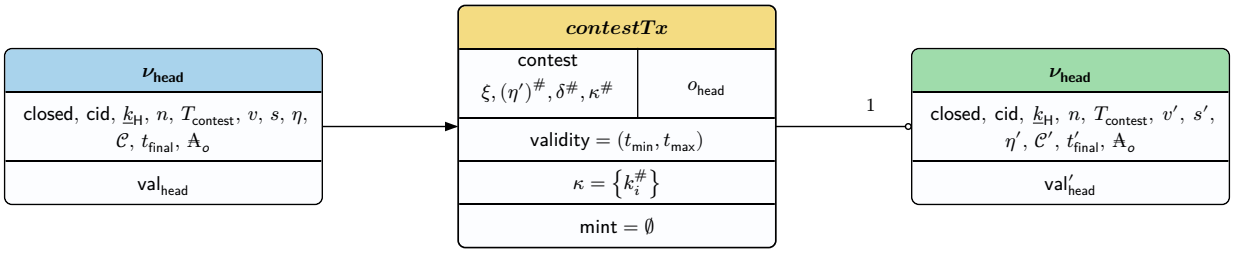

Figure 9: *contest* transaction spending the closed head output and producing a different closed head output.

5.8 Fan-Out Transaction

Once the contestation phase is over, a head may be finalized by posting a *fanout* transaction (see Figure 10), which distributes all UTxOs from the head according to the unified accumulator in the closed state. A fanout transaction consists of

- one input spending from ν_{head} holding the ST, and
- outputs $o_1 ... o_m$ to distribute all UTxOs.

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{fanout}, m, \pi, \mathsf{crsRef})$, where:

- m is the number of outputs to distribute from the closed state,
- π is the KZG membership witness,
- crsRef is the output reference of the reference input holding the Common Reference String (CRS).

The validator checks:

1. State is advanced from $\delta_{\mathsf{head}} \sim \mathsf{closed}$ to terminal state final:

$$(\mathsf{closed}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, n, T_{\mathsf{contest}}, v, s, \eta, \mathcal{C}, t_{\mathsf{final}}, \cancel{\mathbb{A}}_o) \underset{m,\pi,\ \mathsf{crsRef}}{\overset{\mathsf{fanout}}{\longrightarrow}} \mathsf{final}$$

   (the `fanout` rule of `_→(_)_`; a closed datum steps to the terminal state).

2. The CRS reference input named by crsRef carries the *canonical* trusted setup: the hash of its decoded G2-point datum equals the canonical setup hash the validator is parameterised with at compile time,

$$\mathsf{hash}(\mathsf{crsData}) = \mathsf{canonicalCRS}^{\#}$$

   (`InvalidCRSDatum` otherwise; an unresolvable or undecodable reference input rejects with `MissingCRSRefInput`/`MissingCRSDatum`). Without this binding an attacker could supply a substituted powers-of-tau setup with known trapdoor τ and forge membership witnesses for arbitrary outputs — and since fan-out is permissionless after the deadline, that would be direct fund theft. The membership check below runs against this canonical CRS.

3. All m outputs are verified as members of the unified accumulator η using the membership witness π:

$$\mathsf{accVerify}(\eta, \{o_1, ..., o_m\}, \pi) = \mathtt{true}$$

4. Transaction is posted after contestation deadline $t_{\mathsf{min}} > t_{\mathsf{final}}$.

5. All tokens are burnt $|\{\mathsf{cid} \mapsto \cdot \mapsto -1\} \in \mathsf{mint}| = n + 1$.

6. The head input value is fully conserved:

$$\mathsf{val}^{\mathsf{in}}_{\mathsf{head}} = \bigoplus_{i=1}^{m} \mathsf{val}(o_i) \oplus \mathsf{val}_{\mathsf{burned}} \oplus \cancel{\mathbb{A}}_o$$

   where $\cancel{\mathbb{A}}_o$ is the ADA overhead carried from the closed datum, and $\mathsf{val}_{\mathsf{burned}}$ is the value of all burned head tokens.

The fan-out checks - the burn count, membership of the distributed outputs in the accumulator, the deadline, value conservation and the canonical-CRS binding - form the `FanoutValid` bundle.

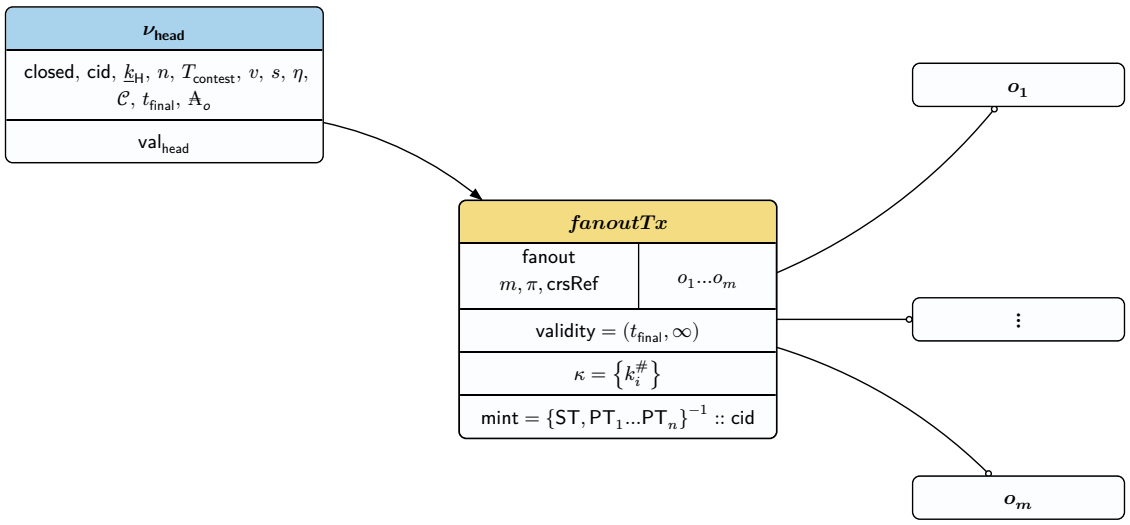

Figure 10: *fanout* transaction spending the closed head output with unified accumulator η and distributing funds with outputs $o_1 ... o_m$.

5.8.1 Intermediate Partial Fan-Out Transaction

When UTxO sets exceed transaction size limits, the protocol distributes UTxOs across multiple transactions using partial fanout. Each intermediate step distributes a subset of UTxOs and transitions the head into the fanoutProgress state, which carries only the fields needed for subsequent steps:

$$(\text{fanoutProgress}, \text{cid}, \underline{k}_{\text{H}}, n, t_{\text{final}}, \eta, \mathbb{A}_o)$$

where n is the number of participants, t_{final} is the contestation deadline (carried from closed), η is the current accumulator commitment, and $\mathbb{A}_o$ is the ADA overhead in the head UTxO not belonging to any L2 UTxO (propagated from the closed datum).

An intermediate partial fanout transaction (see Figure 11) consists of:

- one input spending from ν_{head} in state closed or fanoutProgress, and
- a continuing head output at index 0 in state fanoutProgress with updated accumulator, and
- outputs $o_1 ... o_m$ at indices $1...m$ distributing a subset of UTxOs.

The state-machine validator ν_{head} is spent with $\rho_{\text{head}} = (\text{partialFanout}, m, \text{crsRef})$, where m is the number of UTxO outputs to distribute in this step and crsRef is the output reference of the reference input holding the CRS. The validator checks:

1. $m > 0$ (no zero-output batches).
2. The CRS reference input carries the canonical trusted setup, hash(crsData) = canonicalCRS$^{\#}$ (`InvalidCRSDatum` otherwise; see the fan-out section for the rationale). The exclusion check below runs against this canonical CRS.
3. State transitions into fanoutProgress with updated accumulator. For a closed input:

$$(\text{closed}, \text{cid}, \underline{k}_{\text{H}}, n, T_{\text{contest}}, v, s, \eta, \mathcal{C}, t_{\text{final}}, \mathbb{A}_o) \xrightarrow[m,\ \text{crsRef}]{\text{partialFanoutStart}} (\text{fanoutProgress}, \text{cid}, \underline{k}_{\text{H}}, n, t_{\text{final}}, \eta', \mathbb{A}_o)$$

(the `partialFanoutStart` rule of `_→⟨_⟩_`; closed steps to fanoutProgress). For a fanoutProgress input:

$$(\mathsf{fanoutProgress}, \mathsf{cid}, \underline{k}_\mathsf{H}, n, t_\mathsf{final}, \eta, \mathbb{A}_o) \xrightarrow[m,\ \mathsf{crsRef}]{\mathsf{partialFanoutStep}} (\mathsf{fanoutProgress}, \mathsf{cid}, \underline{k}_\mathsf{H}, n, t_\mathsf{final}, \eta', \mathbb{A}_o)$$

(the `partialFanoutStep` rule of `_→⟨_⟩_`; fanoutProgress steps to itself).

4. The new accumulator η' (from the output datum) is not the G1 generator — all elements have *not* yet been removed (use finalPartialFanout for the last batch):
$$\eta' \neq G_1$$
5. No minting or burning $\mathsf{mint} = \emptyset$.
6. Transaction is posted after contestation deadline $t_\mathsf{min} > t_\mathsf{final}$.
7. Parameters cid, $\underline{k}_\mathsf{H}$, t_final, and $\mathbb{A}_o$ are preserved in the output fanoutProgress datum.
8. Value is conserved: head input value equals head output value plus all distributed outputs:
$$\mathsf{val}^\mathsf{in}_\mathsf{head} = \mathsf{val}^\mathsf{out}_\mathsf{head} \oplus \bigoplus_{i=1}^{m} \mathsf{val}(o_i)$$
9. The new accumulator η' from the output datum correctly represents the remaining UTxOs after removing $S = \{o_1, ..., o_m\}$. The output datum value serves as the exclusion witness:
$$\mathsf{accVerifyExclude}(\eta, S, \eta') = \texttt{true}$$

An intermediate step's conditions form the `PartialFanoutValid` bundle.

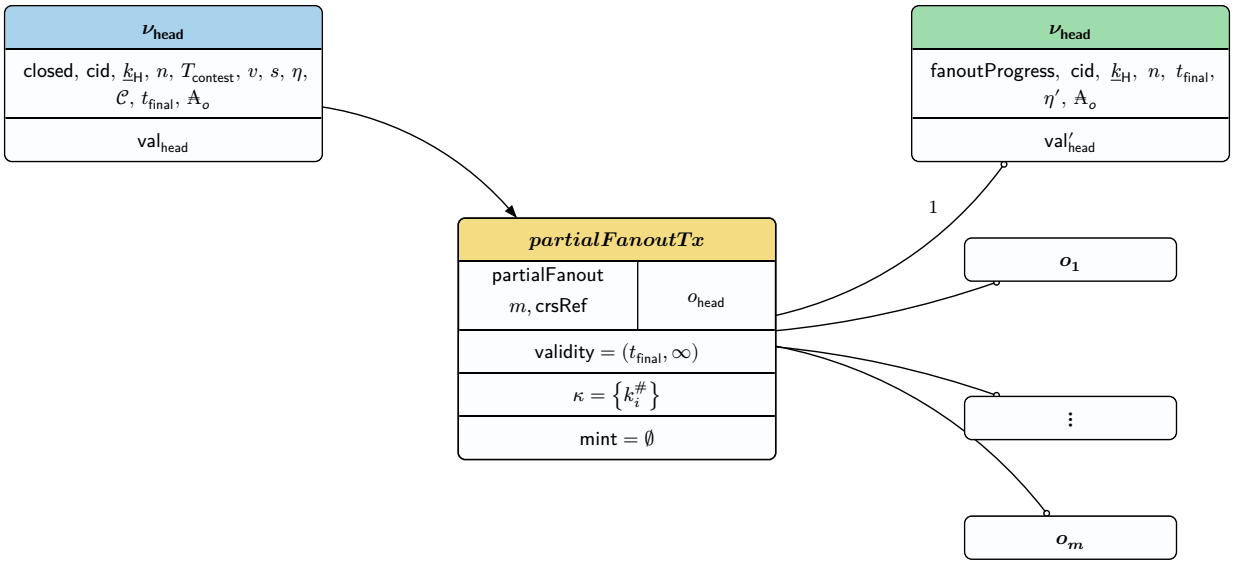

Figure 11: *partialFanout* transaction spending the fanoutProgress head output, distributing outputs $o_1 ... o_m$, and producing a new fanoutProgress head output with updated accumulator η'.

5.8.2 Final Partial Fan-Out Transaction

Once all UTxOs except the last batch have been distributed via partialFanout steps, the final step burns all head tokens and distributes the remaining UTxOs. A final partial fanout transaction (see Figure 12) consists of:

- one input spending from ν_head in state fanoutProgress, and

- outputs $o_1 ... o_m$ distributing the remaining UTxOs (no continuing head output).

The state-machine validator ν_{head} is spent with $\rho_{\mathsf{head}} = (\mathsf{finalPartialFanout}, m, \pi, \mathsf{crsRef})$, where m is the number of UTxO outputs, π is the KZG membership witness, and crsRef is the CRS reference. The validator checks:

1. $m > 0$ (prevents fund theft via a zero-output proof bypass).
2. The CRS reference input carries the canonical trusted setup, $\mathsf{hash}(\mathsf{crsData}) = \mathsf{canonicalCRS}^{\#}$ (`InvalidCRSDatum` otherwise; see the fan-out section for the rationale).
3. State is advanced from fanoutProgress to terminal state final:

$$(\mathsf{fanoutProgress}, \mathsf{cid}, \underline{k}_{\mathsf{H}}, n, t_{\mathsf{final}}, \eta, \mathbb{A}_o) \xrightarrow[m,\pi,\ \mathsf{crsRef}]{\mathsf{finalPartialFanout}} \mathsf{final}$$

   (the `finalPartialFanout` rule of `_→(_)_`; fanoutProgress steps to the terminal state).
4. All head tokens are burnt $|\{\mathsf{cid} \mapsto \cdot \mapsto -1\} \in \mathsf{mint}| = n + 1$.
5. Transaction is posted after contestation deadline $t_{\mathsf{min}} > t_{\mathsf{final}}$.
6. The m distributed outputs are verified as members of the accumulator η using the membership witness π:

$$\mathsf{accVerify}(\eta, \{o_1, ..., o_m\}, \pi) = \texttt{true}$$

7. Value is conserved:

$$\mathsf{val}^{\mathsf{in}}_{\mathsf{head}} = \bigoplus_{i=1}^{m} \mathsf{val}(o_i) \oplus \mathsf{val}_{\mathsf{burned}} \oplus \mathbb{A}_o$$

The final step's conditions form the `FinalPartialFanoutValid` bundle.

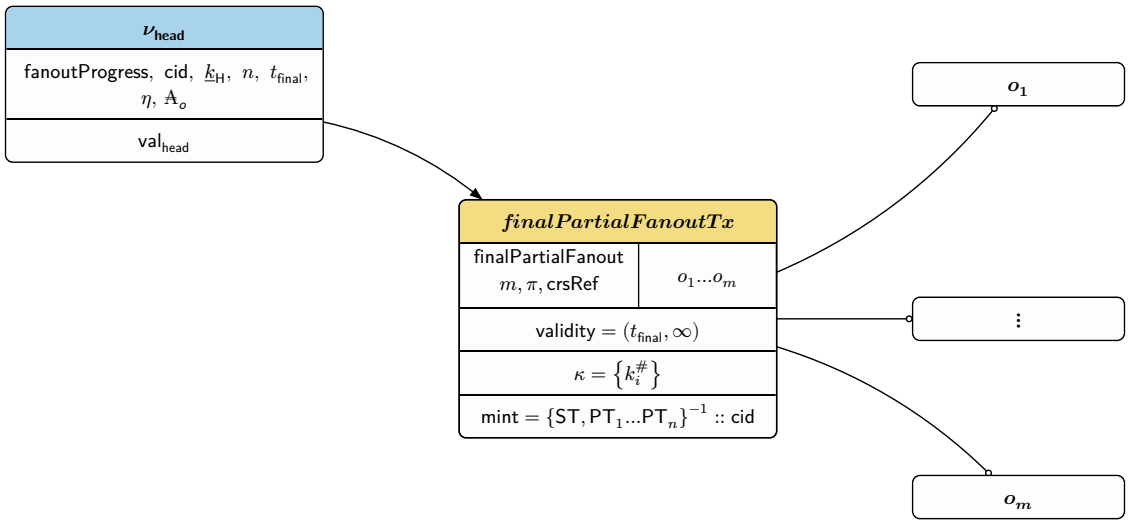

Figure 12: *finalPartialFanout* transaction spending the fanoutProgress head output, distributing the final batch of UTxOs $o_1 ... o_m$, and burning all head tokens to reach final.

The $\mu_{\mathsf{head}}(\varphi_{\mathsf{seed}})$ minting policy governs the burning of tokens via redeemer burn that:

1. All tokens in mint need to be of negative quantity $\forall\{\mathsf{cid} \mapsto \cdot \mapsto q\} \in \mathsf{mint} : q < 0$.

5.9 Machine-checked on-chain properties

The per-transaction conditions of the preceding sections are *soundness* statements: they are only ever consumed in the direction "accepted ⇒ safe". None of them says that a validity bundle is *inhabited* for the states that reach it. That direction matters: an over-strict condition (one stronger than the validator needs) only shrinks the accept set, so it can never falsify a soundness theorem, yet it can strand a reachable head forever (an empty head that cannot be finalised leaves its $n + 1$ state tokens unburnable). This section states the machine-checked properties that close that gap, together with the safety invariants and value-conservation theorems of the on-chain state machine. All of them are proved in the Agda module `OnChainCoverage` (Section 8 shows the statements; the proof terms are typechecked as part of this document's build but not rendered). None of this is temporal liveness (fairness, message delivery, eventual confirmation are out of scope): every statement is an atemporal existence or invariance statement over the transition relation of Section 5.

Two reachability notions ground the statements: `Reachable` is datum-shape reachability over the bare step relation, and `Reachable`$^{\mathsf{v}}$ closes under steps that additionally carry their validity bundles, so it is exactly the set of states a validating chain can occupy. The contest step of `Reachable`$^{\mathsf{v}}$ is stated on concrete datums so the bundle's contester is tied to the datum's newly-appended one.

5.9.1 Non-stuckness (coverage)

Theorem 1 (Empty head is finalisable). *The machine reaches an empty `Closed` head (init opens empty, close-initial keeps it empty), and that reachable state admits a valid $m = 0$ full fan-out: given the context facts (past the deadline, the $n + 1$ tokens burned, value conserved, the canonical CRS reference resolved), the terminal `Fanout` bundle is inhabited, with the 0-output membership witness derived from the accumulator laws (Section 8: `fanout-empty-inhabited`, `finalize-reachable-empty`).*

An `outputsPositive : 0 < m` conjunct on `FanoutValid` would make these lemmas fail to compile (the empty head forces $m = 0$), so the build itself rejects introducing that over-strict guard, which is exactly the defect class this direction exists to catch.

Theorem 2 (Closed heads can finalise). *Any `Closed` head committing to a known UTxO set V ($\eta = \mathsf{accUTxO}(V)$) admits a valid full fan-out of V, provided the transaction actually distributes V; the membership witness is derived from the accumulator laws, so only the genuinely contextual antecedents (deadline, burn, value conservation, pays-out-V, the canonical CRS reference resolved) remain. Reachability is not among them: the witness follows from the accumulator laws for any such datum, so `fanout-coverage` does not take it as a hypothesis. Likewise, a reachable `FanoutProgress` is never stuck: its remaining accumulator is provably non-empty (`progress-nonEmpty`), which* derives *the final batch's $0 < m$ requirement rather than assuming it (Section 8: `fanout-coverage`, `progress-finalizable`).*

5.9.2 Safety invariants of reachable states

The coverage statements above feed the bundle conditions to a constructor, so they are agnostic to the conditions' definitions. The invariants below instead *consume* the structural premises of the transition relation across a reachable run: corrupting a premise breaks the corresponding proof at compile time.

Invariant 1 (No double contest, no resurrection). The contester list of any reachable head is duplicate-free (each contest step's freshness premise $k^{\#} \notin$ contesters is exactly what discharges the induction), and a head that has fanned out is terminal: no transition leaves `Final` (Section 8: `reachable-contesters-distinct`, `final-is-terminal`).

5.9.3 Value conservation (no theft)

Theorem 3 (Every transaction accounts for the head value). *A fan-out (full or final-partial) fully accounts for the head's input value on the ada axis: input ada equals distributed ada plus burned ada plus the carried overhead. Close and contest preserve the head value exactly on both the ada and non-ada axes, and an intermediate partial fan-out splits it exactly between the continuing head output and the distributed batch. Each statement consumes its bundle's value conjunct through the additivity of the value projections (Section 8: `fanout-conserves-ada`, `finalPartialFanout-conserves-ada`, `close-preserves-value`, `contest-preserves-value`, `partialFanout-conserves-ada`).*

Theorem 4 (No output fabrication at fan-out). *The outputs a fan-out transaction actually pays (the anchored `distributedOuts ctx m`, the first m transaction outputs) are a subset of the accumulator-committed set V: consuming the membership conjunct together with the accumulator soundness law, a fan-out cannot distribute an output the head did not commit to (Section 8: `fanout-distributes-committed`, `finalPartialFanout-distributes-committed`).*

A validly-initialised head is moreover a reachable state, tying the μ_{head} init conditions (`versionZero`, `etaEmpty`) to the reachability all of the above is stated over (Section 8: `init-reachable`).

5.9.4 The contest game is bounded

The security section's completeness theorem (Section 7) assumes "the latest multi-signed snapshot wins the close/contest game"; the on-chain half of that game is bounded by two machine-checked facts. First, the contestation deadline of any validly-reachable `Closed` head is at most the close-time deadline plus one contestation period per recorded contester: each contest extends the deadline by at most T, consuming the close and contest deadline equations across the whole run.

Second, the contesters are drawn from the head's n participation-token names, so there are at most n of them. The participant-set facts are taken as *module hypotheses* rather than global postulates on purpose: "a unit quantity of asset $(\mathsf{cid}, k^{\#})$ in a head-output value names one of the head's init-minted participation tokens" is true on-chain by once-only minting (the μ_{head} seed is spent, so policy-cid tokens beyond the $n+1$ minted at init can never exist), but it is not true of the raw value model, where any map literal can carry any asset. Scoping the fact as a hypothesis keeps the theory honest: the theorems hold in any model or run where the values at the head are ledger-produced.

Theorem 5 (The contest window is bounded). *Under the once-only-mint hypotheses, every recorded contester of a validly-reachable `Closed` head is a participant, so (with no-double-contest) at most n contests can ever be recorded, and the contestation deadline is at most $(t_{\mathrm{hi}} + T) + n \cdot T$ for the close transaction's upper validity bound t_{hi}: the deadline moves at most n times, and fan-out is enabled by close plus $n+1$ contestation periods (Section 8: `contesters-are-participants`, `contesters-bounded`, `contest-window-bounded`). This is the atemporal safety half of the contest game; no fairness or liveness is assumed.*

5.9.5 The bridge to the extracted reference checker

The `spec` ⇒ `extracted-reference` half of the differential chain (the Scope note of Section 7) is proved per conjunct in the typecheck-only `ReferenceBridge` module. Its flagship composition lemma is re-stated here so the rendered document carries one representative: a single `closeValid` bundle (shown for the `closeInitial` case) discharges the extracted close checker, the value preservation checker, the no-mint checker and the shared participant-signature checker at once, on the same inputs the `hydra-tx` `HeadValidatorAgreement` suite feeds the real validator. The other transaction families compose identically; their statements live in `ReferenceBridge`, and the injected mocks and encoding postulates the bridge rests on are the drift-checked ledger of Section 8.2.

6 Off-Chain Protocol

This section describes the actual Coordinated Hydra Head protocol, an even more simplified version of the original publication [1]. See the protocol overview in Section 2 for an introduction and notable changes to the original protocol. While the on-chain part already describes the full life-cycle of a Hydra head on-chain, this section completes the picture by defining how the protocol behaves off-chain and notably the relationship between on- and off-chain semantics. Participants of the protocol are also called Hydra head members, parties or simply protocol actors. The protocol is specified as a reactive system that processes three kinds of inputs:

1. On-chain protocol transactions as introduced in Section 5, which are posted to the mainchain and can be observed by all actors:
   - `initialTx`: opens a head
   - `depositTx`: some UTxO was deposited to be incremented
   - `recoverTx`: deposited UTxO was recovered
   - `incrementTx`: adds UTxO to an open head
   - `decrementTx`: removes UTxO from an open head
   - `closeTx`: closes a head
   - `contestTx`: contests a closed head
   - `fanoutTx`: distributes all UTxOs from a closed head in one step
   - `partialFanoutTx`: distributes a subset of UTxOs, transitioning to fanoutProgress (intermediate)
   - `finalPartialFanoutTx`: distributes the last subset and burns tokens (terminal from fanoutProgress)

   Also, a special input when time advanced on chain may be used:
   - `tick`: time advanced on chain
2. Off-chain network messages sent between protocol actors (parties):
   - `reqTx`: to request a transaction to be included in the next snapshot
   - `reqDec`: to request exclusion of UTxO via a decommit transaction
   - `reqSn`: to request a snapshot to be created & signed by every head member
   - `ackSn`: to acknowledge a snapshot by replying with their signatures
3. Commands issued by the participants themselves or on behalf of end-users and clients

- `init`: to open a head
- `close`: to request closure of an open head

The behavior is fully specified in Figure 13, while the following paragraphs introduce notation, explain variables and walk-through the protocol flow.

6.1 Assumptions

On top of the statements of the protocol setup in Section 4, the off-chain protocol logic relies on these assumptions: **TODO:** move/merge with protocol setup?

- Every network message received from a specific party is checked for authentication. An implementation of the specification needs to find a suitable means of authentication, either on the communication channel or for individual messages. Unauthenticated messages must be dropped.
- The head protocol gets correctly (and with completeness) notified about observed transactions on-chain belonging to the respective head instance.
- All inputs are processed to completion, i.e. run-to-completion semantics and no preemption.
- Inputs are deduplicated. That is, any two identical inputs must not lead to multiple invocations of the handling semantics.
- Given the specification, inputs may pile up forever and implementations need to consider these situations (i.e. potential for DoS). A valid reaction to this would be to just drop these inputs. Note that, from a security standpoint, these situations are identical to a non-collaborative peer and closing the head is also a possible reaction.
- The lifecycle of a Hydra head on-chain does not cross (hard fork) protocol update boundaries. Note that these inputs are announced in advance hence it should be possible for implementations to react in such a way as to expedite closing of the head before such a protocol update. This further assumes that the contestation period parameter is picked accordingly.

6.2 Notation

- $U \circ \mathrm{tx}$ applies transaction tx to the UTxO set U, yielding the updated set, or $\perp$ if tx does not apply (a conflict). It is kept abstract as part of the off-chain trust base; see the `applyTxs` laws below.
- **on** *event* specifies how the protocol reacts on a given input *event*. Further information may be available from the constituents of *event* and origin of the input.
- **require** p means that boolean expression $p \in \mathbb{B}$ must be satisfied for the further execution of a routine, while discontinued on $\neg p$. A conservative protocol actor could interpret this as a reason to close the head.
- **wait** p is a non-blocking wait for boolean predicate $p \in \mathbb{B}$ to be satisfied. On $\neg p$, the execution of the routine is stopped, queued, and reactivated at latest when p is satisfied.
- **multicast** *msg* means that a message *msg* is (channel-) authenticated and sent to all participants of this head, including the sender.
- **postTx** tx has a party create transaction tx, potentially from some data, and submit it on-chain. See Section 5 for individual transaction details.

- **output** *event* signals an observation of *event*, which is used in the security definition and proofs of Section 7. This keyword can be ignored when implementing the protocol.

6.3 Variables

Besides parameters agreed in the protocol setup (see Section 4), a party's local state consists of the following variables:

- $\hat{v}$: Last seen open state version.
- $\hat{s}$: Sequence number of latest seen snapshot.
- $\hat{\Sigma} \in (\mathbb{N} \times \mathbb{H})^*$: Accumulator of signatures of the latest seen snapshot, indexed by parties.
- $\hat{\mathcal{L}}$: UTxO set representing the local ledger state resulting from applying $\hat{\mathcal{T}}$ to $\bar{S}.U$ to validate requested transactions.
- $\hat{\mathcal{T}} \in \mathcal{T}^*$: List of transactions applied locally and pending inclusion in a snapshot (if this party is the next leader).
- $\text{tx}_\alpha \in \mathcal{T}$: Pending deposit transaction[1] to be used for incrementing the head state.
- $\text{tx}_\omega \in \mathcal{T}$: Pending decrement transaction, whose outputs are to be withdrawn from the head.
- $\mathcal{D}$: Set of deposit objects tracking deposit transactions with their status.
- $\bar{\mathcal{S}}$: Snapshot object of the latest confirmed snapshot which contains:

$\bar{\mathcal{S}}.v$	snapshot version
$\bar{\mathcal{S}}.s$	snapshot number
$\bar{\mathcal{S}}.\mathcal{T}$	list of transactions relating this snapshot to the previous
$\bar{\mathcal{S}}.U$	snapshotted UTxO set
$\bar{\mathcal{S}}.U_\alpha$	pending UTxO to increment
$\bar{\mathcal{S}}.\varphi_\alpha$	deposit the pending UTxO to increment comes from
$\bar{\mathcal{S}}.U_\omega$	pending UTxO to decrement
$\bar{\mathcal{S}}.(\eta')^\#$	hash of the accumulator commitment over the snapshotted UTxO set

where constructor $\text{snObj}(v, n, T, U, U_\alpha, U_\omega)$ initializes a new snapshot object with $\bar{\mathcal{S}}.(\eta')^\# = \bot$. The Agda `Snapshot` record additionally carries the head id `cid` (the first signed component of the message $\mathsf{cid} \parallel v \parallel s \parallel (\eta')^\# \parallel \delta^\# \parallel \kappa^\#$), the decommit- and commit-output-set hashes $\delta^\#/\kappa^\#$ (the node's `decommitOutputsHash`/`commitOutputsHash`, bound into the signed message), the deposit reference $\bar{\mathcal{S}}.\varphi_\alpha$ the pending increment comes from (bound into $\kappa^\#$, node `depositTxId`) and the aggregate-signature slot $\bar{\mathcal{S}}.\sigma$ ($\bot$ until confirmed), which the figure leaves implicit.

Additionally, deposit objects are created using $\text{depositObj}(U, t_{\text{created}}, t_{\text{deadline}}, \text{status})$ where status can be Inactive, Active, or Expired.

In Agda, a UTxO set is a finite map from output references to outputs, and a party's local state and the protocol messages are captured by dedicated record types.

[1] In fact this would only need to be a transaction id to look up the corresponding deposit in $\mathcal{D}$

The local-ledger application $\circ$ and the UTxO-map operations the handlers use are kept abstract, as the off-chain trust base: `applyTxs` with its nil and compositionality laws (the two ledger laws the Section 7 proofs consume), the joint-applicability predicate `Applicable`, and the map operations recording how $\hat{\mathcal{L}}$ changes.

6.4 Protocol flow

6.4.1 Initializing the head

`init`. Before a head can be initialized, all parties need to exchange and agree on protocol parameters during the protocol setup phase (see Section 4), so we can assume the public Cardano keys $\underline{k}_{\mathsf{C}}^{\text{setup}}$, Hydra keys $\tilde{k}_{\mathsf{H}}^{\text{setup}}$, as well as the contestation period $T_{\mathsf{contest}}^{\text{setup}}$ are available. One of the clients then can start head initialization using the **`init`** command, which will result in an *init* transaction being posted. Not strictly a protocol parameter, but the deposit period T_{deposit} would also be made available from configuration.

`initialTx`. All parties will receive this *init* transaction and validate announced parameters against the pre-agreed *setup* parameters, as well as the structure of the transaction and the minting policy used. This is a vital step to ensure the initialized Head is valid, which cannot be checked completely on-chain (see also Section 5.1). Importantly, parties must verify that the transaction mints the correct tokens (one state thread token and one participation token per head member). This also helps in to distinguish *init* transactions from *increment* or *decrement* transactions, which may produce similar outputs without minting.

Since the *init* transaction directly opens the head with an empty UTxO set, the parties initialize their local state upon observing it: the local ledger state $\hat{\mathcal{L}} = \emptyset$, the seen transaction set $\hat{\mathcal{T}} = \emptyset$, the seen head state version $\hat{v} = 0$, and the snapshot number $\hat{s} = 0$. No deposit transaction $\mathrm{tx}_\alpha = \perp$ and no decrement transaction $\mathrm{tx}_\omega = \perp$ are pending, and the last confirmed snapshot is initialized accordingly $\bar{\mathcal{S}} \leftarrow \mathsf{snObj}(0, 0, [], \emptyset, \emptyset, \emptyset)$.

6.4.2 Processing transactions off-chain

Transactions are announced and captured in so-called snapshots. Parties generate snapshots in a strictly sequential round-robin manner. The party responsible for issuing the i^{th} snapshot is the *leader* of the i^{th} snapshot. Leader selection is round-robin per the $\underline{k}_{\mathsf{H}}$ from the protocol setup. While the frequency of snapshots in the general Head protocol [1] was configurable, the Coordinated Head protocol does specify a snapshot to be created after each transaction.

`reqTx`. Upon receiving request $(\texttt{reqTx}, \mathrm{tx})$, the transaction is applied to the *local* ledger state $\hat{\mathcal{L}} \circ \mathrm{tx}$. If not applicable yet, the protocol does **wait** to retry later or eventually marks this transaction as invalid (see assumption about events piling up). After applying and if there is no current snapshot in flight ($\hat{s} = \bar{\mathcal{S}}.s$) and the receiving party p_i is the next snapshot leader, a message to request snapshot signatures **`reqSn`** is sent.

`reqDec`. Upon receiving request $(\texttt{reqDec}, \mathrm{tx}_\omega)$, the transaction is checked against the *local* ledger state and if it is not applicable yet or another deposit or decommit is pending still, the protocol does **wait** to retry later or eventually marks the decommit as invalid. After applying tx, its outputs are removed from *local* ledger state $\hat{\mathcal{L}}$ so that they are not available any more and the decommit transaction is kept in the local state (tx_ω). If there is no current snapshot in flight $(\hat{s} = \bar{\mathcal{S}}.s)$ and the receiving party p_i is the next snapshot leader, a message to request snapshot signatures **`reqSn`** containing the decrement transaction tx_ω is sent.

`depositTx`. Upon observing a deposit transaction, each party records the deposit index by deposit transaction id to their local deposit registry $\mathcal{D}$ with status Inactive. The deposit contains deposited UTxO U, creation time t_{created}, and deadline t_{deadline}. A transaction is only recognised as a deposit if its inline datum decodes as $\delta_{\mathsf{deposit}}$ and its declared commits total exactly the value locked at ν_{deposit} (the node's deposit observation, `observeDepositTx`, drops anything else) — the observation-time gate that discharges the off-chain obligation of Section 5.2: a malformed deposit never enters $\mathcal{D}$, so no honest party ever signs a snapshot claiming it.

`recoverTx`. Upon observing a recover transaction, each party drops the corresponding entry from its deposit registry $\mathcal{D}$.

`tick`. Whenever time advances (on-chain) to point t, parties update the status of deposits in $\mathcal{D}$ accordingly using the configured deposit period T_{deposit}:

- Expired when deadline passed (or too soon): $t > t_{\text{deadline}} - T_{\mathsf{deposit}}$
- Active when deposit settled enough: $t > t_{\text{created}} + T_{\mathsf{activate}}$

When deposits become Active and no other deposit / decommit is pending, and the party is the next snapshot leader, it may request a new snapshot including the deposit transaction tx_α.

The `tick` deposit-status transition is the function `depositStatusAt`, the off-chain twin of the extracted reference's `depositStatusRef`.

`reqSn`. Upon receiving request $\left(\texttt{reqSn}, v, s, \underline{\mathrm{tx}}_{\mathsf{req}}, \mathrm{tx}_\alpha, \mathrm{tx}_\omega\right)$[2] from party p_j, the receiving p_i **require**s that only a deposit or decommit may be pending, that s is the next snapshot number and that party p_j is responsible for leading its creation. Party p_i may have to **wait** until the previous snapshot is confirmed $(\bar{\mathcal{S}}.s = \hat{s})$ and until v refers to the current open state version. The version check is a **wait**, not a **require**: a follower can receive the request for a bumped version before its own chain handler has observed the transaction that bumped it, so a mismatch parks the request until the versions agree instead of aborting the routine. Furthermore, the protocol validates the snapshot request by:

[2]Snapshot requests with only transaction identifiers and output references are possible if all parties keep an index of previously seen transactions and their identifiers.

1. If a decommit is requested: verify the transaction is applicable to the last confirmed UTxO set and update the active utxo set with it
2. If a decommit is requested: verify it produces at least one output, since *decrement* materializes the decommit outputs and requires one
3. If a deposit is requested: verify the corresponding deposit is Active and update the active utxo set with it
4. If we are on the same version as the last snapshot, any requested decommit or deposit must match the last snapshot.
5. Verify all requested transactions $\underline{\text{tx}}_{\text{req}}$ are applicable to the active UTxO set

Only then, p_i increments their seen-snapshot counter $\hat{s}$, resets the signature accumulator $\hat{\Sigma}$, and computes the UTxO set of the new local snapshot as $U \leftarrow U_{\text{active}} \circ \underline{\text{tx}}_{\text{req}}$. Then, p_i creates a signature σ_i using their signing key $k_{\text{H}}^{\text{sig}}$ on a message comprised by the cid, the new snapshot number $\hat{s}$, the new η resulting from canonically combining U (see Section 5.6 for details), and either η_α or η_ω derived from deposited U_α or decommit transaction tx_ω respectively. Note that η_α covers both U_α and the deposit it comes from, identified by the pending deposit transaction tx_α (recorded in the snapshot as $\bar{\mathcal{S}}.\varphi_\alpha$, and bound on-chain as $txId(\varphi_{\mathsf{deposit}})$, see Section 5.4: a deposit is always the first output of its transaction, so its transaction identifies it). The resulting signature therefore authorizes that one deposit and no other recording the same UTxO. The signature is sent to all head members via message $(\texttt{ackSn}, \hat{s}, \sigma_i)$. Finally, the local ledger state $\hat{\mathcal{L}}$ and pending transaction set $\hat{\mathcal{T}}$ get pruned by re-applying all locally pending transactions $\hat{\mathcal{T}}$ to the just requested snapshot's UTxO set iteratively and ultimately yielding a "pruned" version of $\hat{\mathcal{T}}$ and $\hat{\mathcal{L}}$.

`ackSn`. Upon receiving acknowledgment $\big(\texttt{ackSn}, s, \sigma_j\big)$, all participants **require** that it is from an expected snapshot (either the last seen $\hat{s}$ or $+\,1$), potentially **wait** for the corresponding `reqSn` such that $\hat{s} = s$ and **require** that the signature is not yet included in $\hat{\Sigma}$. They store the received signature in the signature accumulator $\hat{\Sigma}$, and if the signature from each party has been collected, p_i aggregates the multisignature $\tilde{\sigma}$ and **require** it to be valid (constructing the signed message as in `reqSn`). If everything is fine, the snapshot can be considered confirmed by creating the snapshot object $\bar{\mathcal{S}} \leftarrow \mathsf{snObj}\big(\hat{v}, \hat{s}, \hat{\mathcal{T}}, \hat{\mathcal{U}}, U_\alpha, \mathsf{outputs}(\text{tx}_\omega)\big)$ and storing the multi-signature $\tilde{\sigma}$ in it for later reference. In case there is a pending decommit, any participant can now submit a *decrement* transaction by providing the just confirmed snapshot with its accumulator commitment η' for the updated UTxO set. If, however, there was a pending deposit, any participant can now submit an `incrementTx` by providing the confirmed snapshot with its accumulator commitment η' for the updated UTxO set. Lastly, if p_i is the next snapshot leader and there are already transactions to snapshot in $\hat{\mathcal{T}}$, a corresponding `reqSn` is distributed.

The network-message handlers above are transcribed as the relation `_handles_↝_`: one constructor per handler arm, each carrying its `require`/`wait` guards from Figure 13 as premises (the reqTx applicability guard, the reqDec no-in-flight guards, the ackSn no-double-sign and all-signed guards, and the reqSn version/number guards). These are normative: the machine-checked results of Section 7.2 consume exactly these premises. The figure remains the authoritative statement of the full off-chain behaviour; the relations formalise exactly the arms those results consume - the

four network handlers here and the deposit/tick/increment/decrement/initialTx chain observations below - while the close/contest/fanout observations and the leader-multicast effects are not modelled at this level.

`decrementTx`. Upon observing the *decrement* transaction, which removed outputs U from the head, the corresponding pending decrement transaction is cleared and the observed version v is used for future snapshots by setting $\hat{v} \leftarrow v$. Note that the version of the open head state is incremented on each *decrement* transaction as described in Section 5.5.

`incrementTx`. Upon observing the *increment* transaction, which added outputs U to the head, the local ledger state $\hat{\mathcal{L}}$ is extended with the newly added UTxO while the pending increment state U_α is cleared. Also the observed version v is used for future snapshots by setting $\hat{v} = v$. Note that the version of the open head state is incremented on each *increment* transaction as described in Section 5.4

Chain observations are transcribed likewise: `ChainEvent` enumerates the modelled chain observations and `_observes_~_` gives one constructor per handler of the figure - the deposit lifecycle above, the version-bumping increment/decrement observations, and the head-opening `initialTx` of the Initializing-the-head paragraph.

6.4.3 Closing the head

`close`. In order to close a head, a client issues the `close` input which uses the latest confirmed snapshot $\bar{\mathcal{S}}$ to construct the accumulator commitment η' for the closing UTxO state and the certificate ξ using the corresponding multi-signature. With these, the *close* transaction can be constructed and posted. See Section 5.6 for details about this transaction.

`closeTx/contestTx`. When a party observes the head getting closed or contested, the η-state extracted from the *close* or *contest* transaction represents the latest head status that has been aggregated on-chain so far (by a sequence of *close* and *contest* transactions). If the last confirmed (off-chain) snapshot is newer than the observed (on-chain) snapshot number s_c, an updated accumulator commitment η' and certificate ξ are constructed and posted in a *contest* transaction (see Section 5.7).

`fanoutTx`. Upon observing a *fanout* transaction, all UTxOs are distributed and the head transitions to final state.

`partialFanoutTx`. When a head needs to be finalized, the node first attempts to post a single *fanout* transaction distributing all UTxOs at once. If the full UTxO set does not fit within the transaction size or script execution budget, the node falls back to partial fanout: it performs a binary search over batch sizes to find the largest batch that fits within protocol limits, minimising the total number of fanout steps. Concretely, for a remaining set of N UTxOs, it searches $[1, N-1]$ using ceiling-division midpoints (biasing toward larger batches) and selects the largest n for which the transaction satisfies both the size limit and the script execution budget.

A batch of size $n < N$ is posted as an intermediate *partialFanout* transaction, which distributes n UTxOs, updates the accumulator commitment to cover only the remaining $N - n$ UTxOs, and transitions the head to the fanoutProgress state. The procedure repeats: after each *partialFanout* observation the node again attempts to finalize all remaining UTxOs in one step; if they still do not fit, another binary search determines the next batch size. *finalPartialFanout* is used for the last batch when all remaining UTxOs fit in a single transaction, burning all head tokens and transitioning to final.

6.5 Machine-checked handler invariants

Two `require` disciplines of Figure 13 — commit/decommit *exclusivity* (at most one of a pending deposit tx_α or a pending decommit tx_ω in flight, the ReqSn handler's **require** $\mathrm{tx}_\omega = \bot \vee \mathrm{tx}_\alpha = \bot$) and the *version discipline* (the seen open-state version $\hat{v}$ is the confirmed snapshot's version or exactly one above) — are not merely transcribed as handler guards but proved to be maintained by every off-chain step. A step of the handler model is a network-message handling or a chain observation (`_→ᴴ_`), and `Reachableᴴ` closes it reflexively-transitively from a given start state.

Invariant 2 (Handler-model safety disciplines). Along any off-chain run from a state satisfying them (such as the state produced by observing `initialTx`, where both pending slots are empty): (i) at most one of a pending commit (tx_α) or a pending decommit (tx_ω) is ever in flight, the figure's `require` $\mathrm{tx}_\omega = \bot \vee \mathrm{tx}_\alpha = \bot$ discipline (`NoBothInFlight`); and (ii) the seen open-state version $\hat{v}$ is the confirmed snapshot's version or exactly one above (`VersionDiscipline`). The system-level versions over multi-party adversarial executions, `noBothInFlightˢ` and `versionDisciplineˢ`, are stated with the machine-checked results of Section 7.2 (Section 8).

`NoBothInFlight` is the deposit/decommit exclusivity. Its preservation rests on reqDec's de-abstracted `U_α = ∅ ∧ tx_ω = ⊥` guard, which re-establishes the invariant; the increment, decrement and initialTx observations clear a pending slot; every other step leaves both slots untouched. Deleting the `pendingDecrement ≡ nothing` premise on `reqDec-pending` makes the proof fail, which is the adversarial check that the guard carries weight.

Hence the discipline holds throughout any off-chain run whose start state satisfies it.

`VersionDiscipline` makes the close/contest version reuse well-formed: the node reuses the open-state version $\hat{v}$ when posting a contest, which is meaningful only if the confirmed snapshot is at the current or the immediately-prior version (it then posts the $\hat{v}$ or $\hat{v} - 1$ signature). Preservation rests on the on-chain authorize-then-bump rule, captured as the increment/decrement observations' premise that a version bump is signed at the confirmed snapshot's version (so $\hat{v}$ never runs more than one version ahead), while `ackSn-confirm` re-establishes the discipline from its own version premise. This turns an otherwise unchecked runtime assumption of the implementation into a theorem of the off-chain state machine.

6.6 Rollbacks and protocol changes

TODO: Explain why rollbacks are no problem to increment/decrement **TODO:** Write about contestation deadline vs. rollbacks

The overall life-cycle of the Head protocol is driven by on-chain inputs (see introduction of Section 6) which stem from observing transactions on the mainchain. Most blockchains, however, do only provide *eventual* consistency. The consensus algorithm ensures a consistent view of the

history of blocks and transactions between all parties, but this so-called *finality* is only achieved after some time and the local view of the blockchain history may change until that point.

On Cardano with it's Ouroboros consensus algorithm, this means that any local view of the mainchain may not be the longest chain and a node may switch to a longer chain, onto another fork. This other version of the history may not include what was previously observed and hence, any tracking state needs to be updated to this "new reality". Practically, this means that an observer of the blockchain sees a *rollback* followed by rollforwards.

For the Head protocol, this means that chain events like `closeTx` may be observed a second time. Hence, it is crucial, that the local state of the Hydra protocol is kept in sync and also rolled back accordingly to be able to observe and react to these events the right way, e.g. correctly contesting this `closeTx` if need be.

The rollback handling can be specified fully orthogonal on top of the nominal protocol behavior, if the chain provides strictly monotonically increasing points p on each chain event via a new or wrapped `rollforward` event and `rollback` event with the point to which a rollback happened:

`rollforward`. On every chain event that is paired or wrapped in a rollforward event $(\texttt{rollback}, p)$ with point p, protocol participants store their head state indexed by this point in a history Ω of states $\Delta \leftarrow \left(\hat{v}, \hat{s}, \hat{\mathcal{U}}, \hat{\Sigma}, \hat{\mathcal{L}}, \hat{\mathcal{T}}, \bar{\mathcal{S}}\right)$ and $\Omega' = (p, \Delta) \cup \Omega$.

`rollback`. On a rollback $(\texttt{rollback}, p_{rb})$ to point p_{rb}, the corresponding head state Δ need to be retrieved from Ω, with the maximal point $p \leq p_{rb}$, and all entries in Ω with $p > p_{rb}$ get removed.

This will essentially reset the local head state to the right point and allow the protocol to progress through the life-cycle normally. Most stages of the life-cycle are unproblematic if they are rolled back, as long as the protocol logic behaves as in the nominal case.

Since the head is directly opened by the *init* transaction, every rollback of on-chain state is effectively a rollback "past open". However, because the head opens with an empty UTxO set and funds are only added via deposits, the critical concern becomes rollbacks of deposit transactions. The deposit settlement period T_{deposit} (see Section 6) ensures that a deposit is only considered Active — and thus eligible for incrementing into the head — after sufficient time has elapsed since its creation on-chain. This means that by the time a deposit is incremented, its on-chain transaction is sufficiently settled and unlikely to be rolled back, preventing inconsistency between the on-chain and off-chain state.

Figure 13: Head-protocol machine for the *coordinated head* from the perspective of party p_i.

7 Security

TODO: Add security experiment Adversaries:

Active Adversary. An *active adversary* $\mathcal{A}$ has full control over the protocol, i.e., he is fully unrestricted in the above **TODO:** above this section there is no security game security game.

Network Adversary. A *network adversary* $\mathcal{A}_\emptyset$ does not corrupt any head parties, eventually delivers all sent network messages (i.e., does not drop any messages), and does not cause the `close` event. Apart from this restriction, the adversary can act arbitrarily in the above experiment.

Random variables:

- $\hat{S}_i$: the set of transactions tx for which party p_i, *while uncorrupted*, output $(\texttt{seen}, \text{tx})$;
- $\bar{C}_i$: the set of transactions tx for which party p_i, *while uncorrupted*, output $(\texttt{conf}, \text{tx})$;
- $\bar{\Sigma}_i$: latest snapshot (s, U) that party p_i performed *while uncorrupted*: output $(\texttt{snap}, (s, U))$;
- H_{cont}: the set of (at the time) uncorrupted parties who produced ξ upon close/contest request and ξ was applied to correct η; and
- $\mathcal{H}$: the set of parties that remain uncorrupted.

Security conditions / events:

- Consistency (Head): In presence of an active adversary, the following condition holds at any point in time: For all i, j, $U_0 \circ (\bar{C}_i \cup \bar{C}_j) \neq \bot$, i.e., no two uncorrupted parties see conflicting transactions confirmed.
- Oblivious Liveness (Head): Consider any protocol execution in presence of a network adversary wherein the head does not get closed for a sufficiently long period of time, and consider an honest party p_i who enters transaction tx by executing $(\texttt{newTx}, \text{tx})$ *each time after having finished a snapshot.*

  Then the following eventually holds: $\text{tx} \in \bigcap_{i \in [n]} \bar{C}_i \vee \forall i : U_0 \circ (\bar{C}_i \cup \{\text{tx}\}) = \bot$, i.e., every party will observe the transaction confirmed or every party will observe the transaction in conflict with their confirmed transactions.[3]
- Soundness (Chain): In presence of an active adversary, the following condition is satisfied: $\exists \tilde{S} \subseteq \bigcap_{i \in \mathcal{H}} \hat{S}_i : U_{\mathsf{final}} = U_0 \circ \tilde{S} \neq \bot$, i.e., the final UTxO set results from applying a set of transactions to U_0 that have been seen by all honest parties (whereas each such transaction applies conforming to the ledger rules).
- Completeness (Chain): In presence of an active adversary, the following condition holds: For $\tilde{S}$ as above, $\bigcup_{p_i \in H_{\mathsf{cont}}} \bar{C}_i \subseteq \tilde{S}$, i.e., all transactions seen as confirmed by an honest party at the end of the protocol are considered.

Note that the original version of the coordinated head satisfies a stronger version of liveness which is important for the 'user experience' in the protocol:

- Liveness (Head): Consider any protocol execution in presence of a network adversary wherein the head does not get closed for a sufficiently long period of time, and consider an honest party p_i who enters transaction tx by executing $(\texttt{newTx}, \text{tx})$.

[3] In particular, *liveness* expresses that the protocol makes progress under reasonable network conditions if no head parties get corrupted.

Then the following eventually holds: $\mathrm{tx} \in \bigcap_{i \in [n]} \bar{C}_i \vee \forall i : U_0 \circ \left(\bar{C}_i \cup \{\mathrm{tx}\}\right) = \bot$, i.e., every party will observe the transaction confirmed or every party will observe the transaction in conflict with their confirmed transactions.[4]

7.1 Proofs

The security properties are stated over the protocol model below. Three of the four are *machine-checked* in Agda - Consistency (`consistency`), Soundness (`soundness`) and Completeness (`completeness`) - with the safety content *derived* from a signature model (below):

- individual party signatures,
- a snapshot *confirmable* only once *every* party signed it (the coordinated head's full multisignature), and
- honest parties signing only *applicable* snapshots, at most one per number, each extending the signer's own confirmed snapshot.

From these the Agda machine-checks that every honest party's confirmed snapshot is applicable to U_0 (so confirmed sets never conflict), that two confirmations of the same snapshot number coincide, and that confirmed snapshots nest by number (`confirmed-nest`).

`confirm` checks the Section 3.2 aggregate multisignature (`msVfy`); `msgOf` is the snapshot's own serialised content (`snapMsg` of its cid, version, number and η-hash, the Section 6 message cid‖v‖s‖η#), so the verified message depends only on the snapshot's identifying fields rather than being a free token. The binding of a verifying signature to a snapshot is formally carried by `ms-unforgeable`. These are theorems about every *currently*-honest party's confirmed snapshot: the random variables $\hat{S}_i/\bar{C}_i$ are scoped to a party *while uncorrupted*; corruption only shrinks the honest set, and the theorems do not constrain a once-honest-now-corrupt party's confirmed set.

The safety perimeter — the assumptions the proofs rest on — is:

1. the ledger semantics (`applyTxs`);
2. per-signature *unforgeability* (`sigUnforge`, EUF-CMA) plus the aggregation scheme's n-of-n decomposition (`aggSound`), from which the aggregate-level `ms-unforgeable` is *derived*;
3. the honest-signing discipline of `signHonest`, part derived and part assumed: the numbering guard (sign exactly the number one above the signer's own confirmed snapshot, hence at most once per number) is *derived* from the fired `reqSn-sign` handler premise together with the no-in-flight precondition and the invariant's `signNumBound`, while chain-extension, applicability-of-the-delta and only-seen enter as *premises* of the `signHonest` constructor - explicit honest-behaviour assumptions from the protocol flow (Figure 13); and
4. for the on-chain bridge only, that the finalized datum's stored accumulator commits to the off-chain final UTxO (the `ηEq` hypothesis of `reflects`, supplied per finalization), irreducible because νHead authenticates η via the multisignature, not by recomputing it.

The verified aggregate is *system-relative*: `AggVerified sys snap` checks the aggregate `aggSigOf sys snap` built from the signatures the system recorded (`sigs sys`). This keeps the confirmation layer non-vacuous - `AggVerified sys snap` is false where the signatures are absent, yet satisfiable where

[4] In particular, *liveness* expresses that the protocol makes progress under reasonable network conditions if no head parties get corrupted.

every party signed, so a model with genuine confirmations exists. LIVENESS is not yet *stated*: its type is left abstract pending a deferred temporal/fairness layer, so nothing about it is assumed to hold. The prose lemmas further below give the informal arguments these proofs mirror.

Scope (what these proofs do and do not cover). To avoid over-reading the word "unified": these proofs and the on-chain validity bundles of Section 5 (`closeValid`, `incrementValid`, …) are two formalizations with *three deliberate meeting points*:

1. the datum-field accessors (the security model reads the on-chain datum through `OC.snapNum`/`OC.ηOf`/`OC.accUTxO`);
2. the signing message (`snapMsg` is *defined* as the same `cid ‖ v ‖ s ‖ η# ‖ δ# ‖ κ#` concatenation the bundles' `snapshotSigOK` verifies, so the two formalizations meet definitionally at the message); and
3. the certificate corollaries of Section 7.2 (`sig-certifies` and the `*-certified` family consume the bundles' `sigOK` conjuncts, together with the close/contest `etaOK` binding and the deposit side's before-deadline check).

The `finalize` step still admits *any* datum with a matching snapshot number, so no reachability theorem consumes a bundle's value-conservation, deadline or contester checks; those are instead cross-checked against the real Plutus validator by the extracted differential oracle (the `Reference`/`ReferenceBridge` modules), not by these theorems. Two further honesty notes:

- *non-vacuity* (that some confirmation is reachable) is a meta-level model-existence argument, not machine-checked, because `msVfy` is an abstract postulate so no closed term proves `AggVerified`;
- the `ηEq` accumulator-commitment is supplied by the finalizer, not enforced by the model, so `Reflects` is conditional on the finalizer having posted the η it signed.

The νDeposit validator (`deposit.ak`) and the off-chain handlers are likewise not part of any machine-checked theorem here; their decidable conjuncts are covered by the extracted differential layer instead (the Claim arm fully - `claimTxValid→ref` and the claim agreement - the Recover arm modulo its serialisation-hash mock, and the handler guards by the `hydra-node` agreement tests).

The confirmed-snapshot ordering that the safety argument relies on is machine-checked, not a free-standing predicate: `agree` (L1: two honest-certified snapshots of the same number coincide) and `cert-nest` (L2: honest-certified snapshots nest by number), both proved over `Reachable` in Section 8 and consumed by the theorems below.

The properties above quantify over whole multi-party executions in the presence of an adversary, so they are stated over an explicit execution model:

- a ledger-application operation `applyTxs`;
- a global System state recording each party's signatures;
- a single-step relation $\_\longrightarrow^{s}\_$ - an honest party signs an *applicable* snapshot, a corrupt party signs arbitrarily, a party confirms a snapshot whose aggregate multisignature verifies, the adversary corrupts a party; and
- the Reachable closure from an initial system.

A snapshot is Certified once every party signed it, so unforgeability is immediate: a certified snapshot carries the confirmer's own honest signature. The machine-checked invariant then derives:

1. every honest party's confirmed snapshot is applicable to U_0, from the honest "sign only applicable" guard;
2. two certified snapshots of the same number are equal, from the honest "one signature per number" guard; and
3. confirmed snapshots nest by number (`confirmed-nest`), from the honest "extend my own confirmed snapshot" guard plus a gap induction using the previous item.

`confirm` checks the Section 3.2 aggregate multisignature (`AggVerified`/`msVfy`). Beyond the ledger `applyTxs` and the scheme's unforgeability (per-signature `sigUnforge` + the `aggSound` decomposition, from which `ms-unforgeable` is derived), the safety argument relies on the honest-behaviour *premises* of `signHonest` - chain-extension, applicability of the delta, only-seen (see the modelling note below; the numbering guard, by contrast, is derived from the fired handler) - which make the confirmed chain linear and monotone, and, for the on-chain side, on the finalization bridge's accumulator-commitment hypothesis.

The off-chain⇒on-chain link is constructed by `reflects`, from a `finalize` step: the conflict-freedom and snapshot-number conjuncts of `Reflects` are derived, leaving the stored accumulator's commitment to the off-chain UTxO as the single assumed conjunct, supplied per finalization as the explicit hypothesis `ηEq` (a hypothesis rather than a global axiom, since `finalize` admits any matching-number datum). LIVENESS additionally needs a temporal/fairness layer (deferred).

This section states the model and the property statements; the machine-checked results are rendered in Section 7.2, and their *proof terms* (the `invariant` induction and its L1/L2/L3 corollaries, the `consistency`/`soundness`/`completeness` derivations, the once-honest-then-corrupt extension and the `reflects` bridge) live in the companion literate module `Hydra.Protocol.SecurityProofs`, typechecked by the build (imported by `Main`) with the proof bodies not rendered, so the properties remain machine-verified.

Modelling note (honest signing discipline: derived vs. assumed). The `signHonest` move follows the off-chain handler model: an honest party signs by firing the `reqSn-sign` handler (OffChain `_handles_↝_`) with no snapshot in flight ($\hat{s} = \bar{s}$). The four honest-signing guards divide as follows.

The numbering guard is *derived*: the fired handler's premise $s = \bar{s} + 1$ (with $\hat{s} = \bar{s}$) makes the signed snapshot exactly one above the signer's *own* confirmed snapshot, and since signing advances $\hat{s}$, the invariant `signNumBound` bounds every prior signature strictly below the new number, so at-most-one-signature-per-number (`sigDedup`) is proved rather than assumed.

The other three guards are *premises* of the `signHonest` constructor - assumptions about honest behaviour, not derived from the off-chain handlers:

- the snapshot's transactions extend the signer's own confirmed snapshot by a delta (chain-extension);
- the delta applies on top of that confirmed snapshot (applicability); and
- the delta has been observed (only-seen).

From these premises the invariant derives the whole-snapshot facts: applicability to U_0 by ledger compositionality (`applyTxs-compose`) from the party's confirmed-applicability invariant, and only-seen for the whole snapshot from the `sigSeen` invariant. The Section 6 prose specifies this regime operationally (round-robin snapshot leader, $s = \hat{s} + 1$, the `reqSn` 'wait' guards); the derived numbering guard and the honest-behaviour premises together make the confirmed chain linear (`agree`) and monotone (`confirmed-nest`).

7.1.1 The system model

A party's confirmed transactions and number are read off its local state; the global `System` records each party's local state, honesty flag, recorded signatures and seen sets; and `Certified` is the n-of-n signing predicate the proofs reason with.

7.1.2 The signing message and unforgeability

The signing message is the Section 6 serialisation, defined as the same concatenation the on-chain signature conjuncts verify; aggregate unforgeability is derived from per-signature EUF-CMA plus the aggregation scheme's decomposition (Section 8).

7.1.3 The step relation

The single-step relation captures honest signing (firing the `reqSn-sign` handler), corrupt signing, confirmation against a verifying aggregate, corruption, finalization, observation, and lifted local off-chain steps.

7.1.4 Initial systems, reachability and the invariant

An initial system has no signatures and genesis confirmed snapshots; `Reachable` closes the step relation from an initial system; and `Inv` is the eight-field invariant carried through every reachable system, proved by the `invariant` induction of Section 7.2 (Section 8).

7.1.5 The property statements

The properties above are stated as types; their proofs are the machine-checked results of Section 7.2 (Section 8).

Consistency.

Lemma 6 (Consistency). *The coordinated head protocol satisfies the* Consistency *property.*

Proof. Observe that $\bar{C}_i \cup \bar{C}_j \subseteq \hat{S}_i$ since no transaction can be confirmed without every honest party signing off on it. Since parties do not sign conflicting transactions (see `reqSn`, 'wait'), we have $U_0 \circ \bar{C}_i \neq \bot$, $U_0 \circ \bar{C}_j \neq \bot$, and $U_0 \circ \hat{S}_i \neq \bot$. Thus, since $\bar{C}_i \cup \bar{C}_j \subseteq \hat{S}_i$ it follows that $U_0 \circ (\bar{C}_i \cup \bar{C}_j) \neq \bot$

*Machine-checked as `consistency` (Section 7.2), **derived** from the signature model: each honest party's confirmed set is applicable (`conf-applicable`: a confirmed snapshot is certified, so it carries that party's own signature, and honest parties sign only applicable snapshots), and the two confirmed sets nest (`confirmed-nest`, derived via `cert-nest` from the honest extend-your-own-confirmed guard + agreement). The only safety assumptions are the ledger and the Section 3.2 multisignature's unforgeability (`ms-unforgeable`). The statement also covers parties corrupted after confirming (`consistency-uncorrupted`): since `confirm` requires a multisignature regardless of the confirmer's honesty, every confirmed snapshot is certified (`confCert-all`), so any party's confirmed*

set — including a once-honest party's, the one an on-chain close could be built against — stays consistent with every other's, given at least one honest party. □

Oblivious Liveness. For all lemmas towards oblivious liveness, we assume the presence of a network adversary, and that the head does not get closed for a sufficiently long period of time. We call this the *liveness condition*.

Lemma 7. *Under the liveness condition, any snapshot issued as* $(\texttt{reqSn}, s, T)$ *will eventually be confirmed in the sense that every party holds a valid mulisignature on it.*

Proof. Consider a party p_i receiving message $(\texttt{reqSn}, s, T)$. We demonstrate that p_i executes the code past the 'wait' instruction of the `reqSn` routine.

- Passing the 'require' guard:

Note that the snapshot leader sends the request only if $\hat{s} = \bar{s}$, and for $s = \hat{s} + 1$. Thus, $\hat{s}_i = \hat{s}$ since p_i has already signed the snapshot for $\hat{s}$. The 'require' guard is thus satisfied for p_i.

- Passing the 'wait' guard:

Since the snapshot leader sees $\hat{s} = \bar{s}$, also p_i will eventually see $\hat{s}_i = \bar{s}_i$. Furthermore, since all leaders are honest, it holds that $\hat{\mathcal{U}} \circ \mathcal{T}_{\text{res}} \neq \bot$ by construction.

This implies that every party will eventually sign and acknowledge the newly created snapshot. Finally, the 'require' and 'wait' guards of the `ackSn` code will be passed by every party since an `ackSn` for snapshot number s can only be received for $s \in \{\hat{s}, \hat{s}+1\}$ as an acknowledgement can only be received for the current snapshot being worked on by p_i or a snapshot that is one step ahead — implying that everybody will hold a valid multisignature on the snapshot in consideration. □

Lemma 8 (Eternal snapshot confirmation). *Under the liveness condition, as long as new transactions are issued, for any* $k > 0$*, every party eventually confirms a snapshot with sequence number* $s = k$*.*

Proof. By Lemma 7, any requested snapshot eventually gets confirmed, implying that the next leader observes $\hat{s} = \bar{s}$ and thus, in turn, issues a new snapshot. Thus, for any k, a snapshot is eventually confirmed. □

Lemma 9 (Oblivious Liveness). *The coordinated head protocol satisfies the* Oblivious Liveness *property.*

Proof. Consider the first point in time where a transaction tx enters the system by some party p_i issuing $(\texttt{newTx}, \text{tx})$, and consider the next point in time t when p_i issues a snapshot.

By Lemma 8, this snapshot will eventually be issued and confirmed by all parties.

Let $\hat{\mathcal{T}}$ be the transactions to be considered by p_i's snapshot: $\hat{\mathcal{L}} = \bar{\mathcal{U}} \circ \hat{\mathcal{T}}$ where $\bar{\mathcal{U}}$ is the snapshot prior to p_i's. Since p_i issues $(\texttt{reqTx}, \text{tx})$ after each snapshot, we have that, either,

- $\text{tx} \in \hat{\mathcal{T}}$, in which case $\text{tx} \in \bigcap_{i \in [n]} \bar{C}_i$ after everybody has completed this snapshot, or,
- $\text{tx} \notin \hat{\mathcal{T}}$, in which case $\hat{\mathcal{L}} \circ \text{tx} = \bot$ (tx is still in the wait queue of $(\texttt{reqTx}, \text{tx})$. After everybody has completed this snapshot, it thus holds that $\forall i : U_0 \circ \bar{C}_i = \hat{\mathcal{L}}$, and thus, that $\forall i : U_0 \circ \left(\bar{C}_i \cup \{\text{tx}\}\right) = \bot$.

In both cases, the lemma follows. □

Soundness and completeness.

Lemma 10 (Soundness). *The basic head protocol satisfies the* SOUNDNESS *property.*

Proof. Let T be the set of transactions such that $U_{\mathsf{final}} = U_0 \circ T$. Since U_{final} is multi-signed, it holds that $T \subseteq \hat{S}_i$ (T is *seen*) by every honest party in the head. Furthermore, since honest signatures are only issued for valid transaction, $U_{\mathsf{final}} \neq \bot$ (i.e., U_{final} is a valid state), and soundness follows.

Machine-checked as `soundness` *in Section 7.2 ($U_{\mathsf{final}} = U_0 \circ \tilde{T} \neq \bot$ with $\tilde{T}$ the certified finalized snapshot). The $\neq \bot$ is* ***derived****: a certified snapshot carries an honest party's signature, and honest parties sign only applicable snapshots (*`cert-applicable`*). The $\tilde{T} \subseteq \bigcap_{i \in \mathcal{H}} \hat{S}_i$ conjunct is machine-checked too: an honest party signs only transactions it has observed (the* `signHonest` *seen guard), so a certified snapshot's transactions lie in every honest party's seen set (*`sigSeen-inv`*), proved as the second conjunct of* `soundness`*.* □

Lemma 11 (Completeness). *The basic head protocol satisfies the* COMPLETENESS *property.*

Proof. Consider all parties $p_i \in H_{\mathsf{cont}}$. Since the close/contest process finally accepts the latest multi-signed snapshot, it holds that $U_{\mathsf{final}}.s \geq \max_{p_i \in H_{\mathsf{cont}}}(\bar{s}_i)$, and thus that $\bigcup_{p_i \in H_{\mathsf{cont}}} \bar{C}_i \subseteq \bigcap_{p_i \in \mathcal{H}} \hat{S}_i$, and completeness follows.

Machine-checked as `completeness` *in Section 7.2: each honest party's $\bar{C}_i \subseteq \tilde{T}$ (the finalized snapshot itself) whenever $\bar{s}_i \leq s_f$,* ***derived*** *from* `cert-nest` *(L2),* `confCert-of` *(the party's confirmed snapshot is genesis-or-certified) and* `ms-unforgeable` *(the finalized snapshot is certified). The $\bar{s}_i \leq s_f$ premise is the "latest multi-signed snapshot wins" fact of the close/contest process; our* `finalize` *admits any certified snapshot, so it is a per-party premise rather than derived. The sibling honest-to-honest nesting is* `confirmed-nest`*.* □

7.2 Machine-checked results

The properties of the preceding section are not only stated over the execution model; they are proved. This section renders the machine-checked results: each statement below is the actual Agda type of a theorem in the module `SecurityProofs`, shown here and in Section 8, while the proof terms — and the supporting lemmas and corollaries the prose cites by name — are typechecked as part of this document's build but not rendered. All results are inductions over the `Reachable` closure of the step relation `_→ˢ_` (Section 7); none of them is temporal (liveness remains out of scope). The trust base is exactly the one enumerated in the Proofs preamble: the ledger postulates (`applyTxs` and its laws), per-signature unforgeability plus the aggregation scheme's decomposition (from which `ms-unforgeable` is derived), the honest-behaviour premises of `signHonest`, and, where a result consumes an on-chain validity bundle, the per-instance signature-trust hypotheses noted with that result.

The workhorse is a single induction: every reachable system satisfies the eight-field invariant `Inv` of Section 7 (Section 8: `invariant`). The safety content is derived from the structure of the steps rather than assumed: `confApp` (L3) is discharged at `confirm` from `sigApp` (a certified snapshot carries the honest confirmer's own signature, and honest signatures are only on applicable snapshots), and `sigChain` records, for every honest signature, an extending certified-or-genesis predecessor

(from the `signHonest` premises together with `confCert`), which is what the nesting lemma below consumes. Corruption only ever shrinks the honest set, and `sigs` only grows, so certification facts are carried forward across steps. All of the L1/L2/L3 consequences used by the theorems of this section are projections of this one invariant.

Agreement (L1) is the projection of the invariant's one-signature-per-number field: two certified snapshots of the same number are equal, witnessed by any honest party, who by `Certified` signed both (Section 8: `agree`).

Certified snapshots nest by number (L2). The proof is an induction on the gap between the two numbers: at gap zero the snapshots are equal by `agree`; otherwise the higher snapshot has, by the invariant's `sigChain`, an extending certified-or-genesis predecessor one number below it, so the induction recurses to the predecessor and composes with the extension, the genesis case being impossible for a certified (hence positive-numbered) snapshot (Section 8: `cert-nest`).

The Section 7 nesting obligation follows: two honest parties' confirmed snapshots nest by number, since an honest party's confirmed snapshot is the genesis (whose transaction list is empty, hence trivially contained) or is certified, in which case `cert-nest` applies (Section 8: `confirmed-nest`).

7.2.1 Consistency

Theorem 12 (Consistency). *In any reachable system, the confirmed transaction sets of two honest parties nest (one contains the other) and each is applicable to U_0, so their union never fails to apply; the union form is machine-checked as a single transaction set containing both that is applicable to U_0 (`consistency-union`), making the paper's $U_0 \circ (\bar{C}_i \cup \bar{C}_j) \neq \bot$ literal. The property moreover extends to once-honest-then-corrupt parties:* every *party's confirmed snapshot is genesis-or-certified unconditionally (`confCert-all`), because `confirm` demands a verifying aggregate multisignature regardless of the confirmer's honesty, so* any *two parties' confirmed sets, including one a party confirmed before (or even adopted after) being corrupted, nest and are applicable, given at least one honest witness (`consistency-uncorrupted`) (Section 8).*

The union form: there is a single transaction set `T` containing both honest confirmed sets (their union, i.e. the inclusion-larger of the two, since they nest) that is applicable to U_0.

The Section 7 random variables $\bar{C}_i$ scope a party's confirmed set to a party *while uncorrupted*, and an on-chain close could be built against the confirmed snapshot of a party corrupted after confirming. The extension rests on the fact that certification is unconditional: the only step that changes a party's confirmed snapshot is `confirm`, which requires a verifying aggregate multisignature (hence, by unforgeability, a certificate) whatever the confirmer's honesty flag. This is in fact stronger than the literal Section 7 scoping, since it also covers any snapshot a corrupt party adopts after corruption.

7.2.2 Soundness and completeness

Theorem 13 (Soundness). *In any reachable system with at least one honest party, if a snapshot's aggregate multisignature verifies, then applying its transactions to U_0 succeeds (the final UTxO set exists and is conflict-free) and those transactions have been seen by* every *honest party ($\tilde{T} \subseteq \bigcap_{j \in \mathcal{H}} \hat{S}_j$) (Section 8: `soundness`).*

Both conjuncts are derived: `ms-unforgeable` makes the verified snapshot certified, an honest signer signs only applicable snapshots (`cert-applicable`, giving conflict-freedom), and only transactions it has seen (the `sigSeen` component of the invariant, giving the intersection-seen subset).

Theorem 14 (Completeness). *Every honest party's confirmed transactions are contained in the finalized snapshot (the snapshot whose aggregate multisignature verifies), whenever that party's confirmed number is at most the finalized snapshot's number (Section 8: `completeness`).*

The finalized snapshot is certified (`ms-unforgeable`); the party's confirmed snapshot is genesis (trivially contained) or certified (`confCert-of`), and two certified snapshots nest by number (`cert-nest`), using the party itself as the honest witness. The `confirmedNo ≤ number` premise is the "latest multi-signed snapshot wins the close/contest game" fact of Section 5.6/Section 5.7: the real close/contest process always settles on the latest multi-signed snapshot, whereas the model's `finalize` admits *any* certified snapshot, so the fact enters as a per-party premise rather than being derived.

7.2.3 The on-chain reflection bridge

Theorem 15 (On-chain settlement reflects the off-chain state). *A finalization against a snapshot whose aggregate multisignature verifies yields `Reflects`: the off-chain final UTxO exists (from Theorem 13), the on-chain snapshot number matches (the `finalize` witness), and the stored accumulator commits to that UTxO. Conversely, when the datum reflects a finalized snapshot, the outputs a valid on-chain fan-out actually distributes are a subset of that off-chain final UTxO's outputs (`reflect-sound`, `reflect-fanout-⊆`) (Section 8).*

Two of the three `Reflects` conjuncts are derived; the accumulator commitment is supplied as the explicit per-finalization hypothesis `ηEq`, the irreducible signature-trust assumption: ν_{head} authenticates η via the multisignature over $\mathsf{cid} \parallel v \parallel s \parallel \eta^{\#} \parallel \delta^{\#} \parallel \kappa^{\#}$, not by recomputing $\mathsf{accUTxO}(U)$. It is a hypothesis rather than a global postulate on purpose, since `finalize` admits any datum with a matching snapshot number, a global axiom would have no model, and the finalizer discharges `ηEq` from the η it actually committed.

The fan-out half consumes the bundle's membership conjunct through the accumulator soundness law: the outputs the transaction pays (the anchored `distributedOuts`) are a subset of the outputs of the off-chain final UTxO, tying the transaction's own outputs, not a free set parameter, to the Soundness UTxO.

7.2.4 No settlement without unanimity

Theorem 16 (No settlement without unanimity). *Every Section 5.4–Section 5.7 validity bundle carries a* `sigOK` *conjunct bottoming out in* `snapshotSigOK` *(*MS-Verify *over* cid $\| v \| s \| \eta^{\#} \| \delta^{\#} \| \kappa^{\#}$*), and the model's signing message* `snapMsg` *is* defined *as that same concatenation. Once the snapshot's identifying fields match the signed ones and the redeemer's aggregate signature is the system-recorded one (the per-instance signature-trust hypothesis* `ξEq`*, the same pattern as the reflection bridge's* `ηEq`*), the on-chain conjunct is exactly* `AggVerified`*, and unforgeability certifies: every party signed. Hence no increment, decrement, close or contest settles without a unanimous certificate (*`sig-certifies` *and the per-transaction* `*-certified` *corollaries); the certified snapshot's accumulator hash is the hash of the accumulator actually stored in the produced datum (*`close-η-reflected`*,* `contest-η-reflected`*); and a valid deposit-claim transaction is both unanimously certified and posted before the deposit's recover deadline (*`claimTx-certified`*) (Section 8).*

These corollaries consume the `sigOK` fields of the on-chain bundles, so the off-chain certificate layer and the on-chain validity bundles meet at the signature conjuncts, not only at datum accessors. The field-matching equalities are dischargeable by `refl` for the snapshot actually signed; `ξEq` is per-instance for the same reason `ηEq` is.

The per-transaction corollaries (`increment-certified`, `decrement-certified`, `close-certified`, `contest-certified`) each consume their bundle's `sigOK` field through a one-line application of `sig-certifies` (typechecked, not rendered); the `closeUsed`/`contestUsed`/`closeAny` redeemer variants follow identically (their `sigOK` reduces to `snapshotSigOK` at version $v-1$ or v).

On top of the certificate, close and contest cannot verify a signature over one accumulator while storing another: the `etaOK` conjunct binds the redeemer's $\eta^{\#}$ to the hash of the accumulator stored in the produced datum (the two-line corollaries `close-η-reflected` / `contest-η-reflected`, typechecked but not rendered).

A valid deposit claim (the joint `ClaimTxValid` encoding, ν_{head} increment and ν_{deposit} claim both passing) is unanimously certified *and* posted before the deposit's recover deadline: a deposit can be absorbed into the head neither without every party's signature nor after it has become recoverable.

7.2.5 Handler invariants across adversarial executions

Invariant 3 (No commit/decommit overlap, version discipline (system level)). Throughout any reachable adversarial execution, for every party: a pending commit (tx_{α}) and a pending decommit (tx_{ω}) are never both in flight (`noBothInFlight`^{s}), and the party's seen open-state version is its confirmed snapshot's version or exactly one above (`versionDiscipline`^{s}). Both are seeded by the `Initial` predicate and preserved by every `_→`^{s}`_` step, reusing the local handler-model preservation lemmas of Section 6.5 on the `offChain` step's embedded handler witness (Section 8).

This lifts the `require` disciplines of Figure 13 from the single-party handler model to the multi-party adversarial system: the deposit/decommit exclusivity and the version discipline are properties of every reachable execution, not runtime assertions an honest node merely hopes to maintain.

7.3 Solvency

The per-transaction validity bundles of Section 5 each conserve value locally, and the security results of Section 7.2 establish that every settled snapshot is unanimously certified. Neither layer

states the property that actually protects funds: that the head output's value *covers* the UTxO set the accumulator commits to, so that fanout can distribute what L2 accounting says participants own. The deposit-binding vulnerability violated exactly this while satisfying every local check. This section states the invariant, proves it inductively over the head's on-chain transitions, and derives the payoff at fanout: every distributed output is one the committed L2 set actually contains, funded by the head value the invariant accounts for.

The L2 UTxO set is not recoverable from the on-chain state (the datum carries only the commitment η), so the induction carries it as *ghost state*: a set U alongside the datum, with the two-part invariant

$$\eta = \mathsf{accUTxO}(U) \quad \text{and} \quad \mathsf{val}_{\mathsf{head}} = r_0 \oplus \Sigma_{o \in U}\ \mathsf{val}(o),$$

where r_0 is the value the head output carried at init (the $\mathbb{A}_o$ overhead plus the state and participation tokens, preserved by every transition) and Σ is the value of a UTxO set. The value function and its empty-set law are the only new postulates besides two digest constants; every other ingredient is a named hypothesis with a clear owner, gathered in the `Assumptions` record or supplied per step in the reachability relation below.

What honest signing guarantees about a certified snapshot, in the on-chain vocabulary the induction consumes. A certificate is n-of-n (Theorem 16), so at least one honest party enforced the §6 handler `requires` before signing; these fields are that party's obligations. The pending shape is a *sum*: a snapshot carries a pending commit, a pending decommit, or neither - never both. That is the machine-checked `NoBothInFlight` state invariant of Section 6.5 lifted to the snapshot level, and it is load-bearing below: without it a decrement could settle an increment-shaped snapshot, adopting an accumulator that counts deposited UTxOs no transaction absorbed.

The assumption bundle, relative to a system of parties. The three digest hypotheses are information-theoretic *idealizations* of second-preimage resistance on the digest shapes the induction inverts - stated as injectivity on those shapes, which is stronger than the computational property, exactly as the model's multisignature unforgeability postulates are. The computational reading is the standard reduction: a violation of solvency under the other hypotheses yields a concrete digest collision. They are not global injectivity of `hash` (which would be inconsistent with compression); they apply only at the pair, commitment and output-list shapes actually signed. Owner: crypto. `honest-certified` is the ≥1-honest-signer argument as a *named, consumed* assumption rather than prose: an n-of-n certificate (Theorem 16) contains at least one honest party, whose §6 handler `requires` enforced exactly the `HonestFacts` fields before signing. Owner: the honest-majority premise of Section 7.

Per-step hypotheses, dispatched on the pending shape (trivial for the shapes a step contradicts). `ObservedDeposit` is ledger resolution faithfulness at the observed deposit: any input of *this* transaction whose out-ref names output 0 of the deposit transaction the parties observed resolves to that deposit output, whose value is the recorded `incVal` (owners: the ledger for resolution, the honest `observeDepositTx` for the value). `IncCoherent` / `DecCoherent` relate consecutive committed sets by the settled delta; they package L2 value coherence between settlements and are stated per step. Their reading is over L1-originating value (see `sumValue`): L2-minted assets are outside the accounting and cannot cross the L1 boundary anyway, while an L2 *burn* of an L1-originating asset

would leave the head strictly richer than the committed set - safe for coverage, but breaking the equality form, so the hypotheses additionally assume the L2 traffic between settlements burns no L1-originating value. Deriving them from the off-chain ledger laws is future work alongside the closed-head cases.

The ghost-state reachability relation. Each step takes the on-chain validity bundle, the snapshot's honest facts, the unforgeability-derived digest equalities (the per-instance idiom of the `*-certified` corollaries of Section 7.2), and the per-step ledger hypotheses; the `headValueIn ctx ≡ w` premise is L1 UTxO continuity - the head output this step spends is the one the previous step produced (owner: the ledger). Certification enters through the derived `*-certified` step forms inside `Invariant` below, which consume a `Certified` certificate through the named `honest-certified` assumption instead of taking the honest facts on faith.

Which transitions the relation reaches is a real limit on what the theorem says, and nothing about adding a transition to Section 5 forces a step to be added here - the theorem would simply say less. So the correspondence is enumerated and gated rather than left to inspection: `check-trust-ledger.sh` fails unless every `*Valid` bundle declared in Section 5 is either consumed by a step below or listed there with the reason it is out of reach. Six of the twelve are consumed; the batched fan-out arms are out because value leaves the head across several transactions, which the single-step shape does not express, and the νDeposit arms because they govern the deposit UTxO rather than the head output whose value this invariant tracks.

The invariant. The increment case is where the deposit-binding fix is consumed: the digest equality inverts (`κ#-pair-inj`) *only because* `depositCommitsHashOf` binds the transaction id, the first-output rule pins the claimed out-ref to the observed deposit output, and `IncrementValid.onlyClaimedDeposit` equates the absorbed value with that one deposit's. Under the pre-fix digest the chain from "the signature verifies" to "the head absorbed the recorded value" breaks at the first link.

All three of those are *fields of the validity bundle* the step already takes, not hypotheses this relation grants itself. That is what makes the dependency load-bearing rather than asserted: drop the anti-siphon conjunct from `IncrementValid`, or weaken `depositCommitsHashOf` back to hashing the datum alone, and this proof stops typechecking. The decrement case consumes `decommitNonEmpty` and the pending-shape sum: an increment-shaped snapshot presented to a decrement forces the materialized output list to hash like the empty list, contradicting the at-least-one rule.

The certified step forms: the same steps, taking a `Certified` certificate and deriving the honest facts through the named `honest-certified` assumption. These are the intended entry points - a real chain history enters the relation through a certificate, never through free-floating honest facts.

The payoff. `membersOK` alone says the distributed outputs are members of *whatever* the stored accumulator commits to; only together with the invariant does that become "members of the L2 set the parties actually built". Combined with the fanout value equation, a solvent head can neither fan out phantom outputs nor come up short funding the real ones.

Two companions round the section off. `SolvencyCounterModel` (typecheck-only, not rendered) keeps the pre-fix attack derivable in miniature: against a digest covering datum content alone, a copied-datum deposit holding less value is accepted under the real deposit's signature and leaves the head

insolvent, while the id-binding digest rejects the same claim by identity - regression documentation the checker rebuilds on every build. Not yet covered here: the `closeUsed`/`contestUsed` redeemers (their stored accumulator combines a pending delta, needing a generalized commitment invariant), partial fanout, and deriving the per-step L2 value-preservation hypotheses from the off-chain ledger laws.

8 Agda formalisation

The specification above is *literate Agda*: every definition, validity bundle, and proof is machine-checked by Agda (`agda Main.lagda.typ`) as part of building this document. To keep the body readable, the rendered Agda is collected here rather than shown inline; each block appears under the section it supports, in document order, and the body links here in place. The appendix is deliberately restricted to the *machine-checked theorem statements* (the coverage, safety, value-conservation and security results of Section 5.9, Section 6.5 and Section 7.2, and the solvency invariant of Section 7.3) together with the property types they inhabit. Everything else — the datum/redeemer types, the transition relations, the validity bundles, the helper definitions, every `postulate`, and all proof bodies — is typechecked as part of the build but not rendered, as are the typecheck-only modules `Prelude`, `Reference`, `ReferenceBridge`, `RefReflection`, `OffChainReference` and `SolvencyCounterModel`. Section 8.2 lists what the formalisation takes on trust.

8.1 Reading the Agda (for Haskell programmers)

The definitions, transition rules and security proofs in this document are real Agda code, type-checked as part of the build (`nix build .#spec`); the prose and math render alongside them. If you know Haskell but are new to Agda, this short glossary maps the Agda idioms you will meet to their Haskell intuition.

Assumptions vs. proofs. The single most important thing to know: a block introduced by the Agda keyword `postulate` is an *assumption* (an axiom the spec takes on trust, such as the ledger semantics or the cryptographic unforgeability of signatures), *not* something proved. Everything else, given by a defining equation, is a definition or a type-checked proof. Postulate blocks are not rendered in this document (only theorem statements are); the full inventory lives in Section 8.2. Postulates are also not the only assumptions: some enter as module *hypotheses* or constructor *premises* (the honest-behaviour premises of `signHonest` in Section 7, the `ContestBound` hypotheses of Section 5.9, per-instance hypotheses like `ηEq`). Section 8.2 collects all of these in one place; auditing the sources means searching for `postulate` *and* reading the flagged premises.

Types, values and proofs. Agda is dependently typed: types may mention values. `Set` is the type of (small) types, Haskell's kind `Type`. A function type `(x : A) → B x` is a *dependent* function whose result type may depend on the argument; `∀ {x} → B` is the same with `x` an *implicit* argument Agda infers, like an inferred `forall`. Propositions are types and a proof is a value of that type, so a function `P → Q` is read both as "a function" and as "P implies Q".

Logic and data. `_×_` is a pair, read as logical *and*; `_⊎_` is `Either`, read as *or* (constructors `inj₁`/`inj₂`); `⊥` is the empty type (`False`; `⊥-elim` is "from a contradiction, anything"); `⊤` is unit. `Σ[ x ∈ A ] B` is a dependent pair (a value `x : A` together with a `B`, often read "there exists an `x` such that B"); Agda `record`s are sugar for nested `Σ`.

Equality. `_≡_` is *propositional* equality: `a ≡ b` is the type of *proofs* that `a` and `b` are the same value, distinct from a `Bool`-valued test `a == b`. `refl` proves `a ≡ a`; `sym`/`trans`/`cong` are symmetry, transitivity and congruence; `subst P eq px` rewrites a proof `px : P a` along `eq : a ≡ b` into a `P b`. Where the spec runs a `Bool` equality and needs to turn it into `≡`, the bridge lemma is

named `==-sound` (proved in the typecheck-only `RefReflection` module, so it does not appear in this appendix).

Pattern matching. Definitions are equations over constructors, as in Haskell. `with e` adds `e` as an extra argument to split on, refining the goal by what `e` turned out to be. An *absurd pattern* `()` discharges a case that cannot occur because its type has no constructor (for example, a membership proof in the empty list), with no right-hand side.

Recurring types. `ℕ` the naturals; `List`/`_∷_`/`[]`, `Maybe`/`just`/ `nothing` as in Haskell; `Fin n` the naturals below `n` (a bounded index); `Vec A n` a length-`n` list; `ℙ A` a finite set and `_⇀_` a finite map (from the set-theory library); `_∈ˡ_` list membership and `_⊆ˡ_` list inclusion. Most unicode names are ordinary identifiers, defined in the typecheck-only `Prelude` module; Section 3 is the rendered counterpart for the mathematical notation.

Relations as transitions. The on-chain state machine is an inductively-defined relation `_→⟨_⟩_` (a datum steps to a datum under a redeemer), and the security model uses a step relation `_→ˢ_` with its reflexive-transitive closure `Reachable`. A value of such a type is a *proof* that a particular step (or run) is allowed; the proofs in Section 7 are inductions over these.

Validity bundles. A validator's requirements are written as a `record` (e.g. `CloseValid`) with one named field per checkable condition (`step`, `deadlineOK`, `valuePreserved`, …) - like a Haskell record of proofs. The bundle is inhabited exactly for valid transactions, and a proof reads a condition by its field name (`CloseValid.deadlineOK b`) rather than by tuple position. (The lowercase `closeValid ctx d d' ct` is the predicate that returns this record for well-shaped datums, and is empty otherwise.)

Extraction. The decidable checker in `Reference.agda` is compiled to Haskell by Agda's GHC backend (MAlonzo) and run in the `hydra-tx` test suite as a second oracle against the real Plutus validator (see Section 7).

8.2 What the formalisation assumes

The trust base, in five families:

- **Ledger and crypto primitives** (the typecheck-only `Prelude`, postulates): `hash`, `bytes`/ `concat`, the multisignature verifier `msVfy`, the `Value` algebra (`_+ᵛ_`/`εᵛ` with the associativity and identity laws the proofs use) and its projections (`adaOf`, `nonAdaOf`, `quantityOf`, `stQty`, `headTokenCount`); the off-chain ledger `applyTxs` with its nil and compositionality laws (Section 6).
- **Accumulator laws** (Section 5): `accUTxO`/`accVerify`/`accVerifyExclude` with the specifying laws `accUTxO-∅`, `accVerify-sound`/`-complete`/`-self` and `setSize`; the KZG construction itself is not modelled.
- **On-chain search postulates** (Section 5): `burnedValue`, `burnedCount`, `mintedCount`, `μHead`, `signerKeyHash` - witnesses over the opaque value and key-set models - plus the context lookups the `Context` model does not expose: `depositDatumCommitsHash` (the hash of the commit list decoded from the claimed deposit's datum; the full commit digest `depositCommitsHashOf` is a *definition* over it, binding the deposit's transaction id) and `crsDatumHashAt`/`canonicalCRS#`

(the CRS reference-input datum hash and the canonical trusted-setup constant the fanout bundles bind it to).

- **Security-model assumptions** (Section 7): per-signature EUF-CMA (`sigUnforge`) plus the aggregation scheme's decomposition (`aggSound`), from which `ms-unforgeable` is *derived*; `aggKey`/`aggSigOf`/`PartyVerified` and the glue `outsOf`; the honest-behaviour premises of the `signHonest` constructor; and per-instance hypotheses supplied at each use (the `ηEq` accumulator-commitment hypothesis of `reflects`, the `ξ ≡ aggSigOf` hypothesis of the `*-certified` family). The `ContestBound` module hypotheses of Section 5.9 are of the same kind.
- **Solvency layer** (Section 7.3): the L1-originating set-value fold `sumValue` with its empty-set law and the two no-deposit digest constants (postulates, the latter an encoding of the same family as the bridge's `refCodeOf`); the `Assumptions` record - digest injectivity at the signed shapes (`κ#`/`η#`/`δ#`), an information-theoretic idealization of second-preimage resistance whose computational reading is the standard reduction, plus `honest-certified`, the ≥1-honest-signer argument as a named, consumed assumption; and the per-step hypotheses of `SolventReach` (L1 continuity and observed-deposit resolution owned by the ledger, settled-delta value coherence owned by the L2 ledger rules over L1-originating assets, assuming burn-free traffic between settlements - L2-minted tokens sit outside the accounting and cannot cross the L1 boundary). The `+ᵛ-cancelʳ` cancellation law joins the Prelude Value algebra family above.
- **Bridge layer** (typecheck-only `ReferenceBridge`/`RefReflection`): 6 injected const-true mocks (crypto/accumulator/hash conjuncts the differential covers against the real validator) and 7 encoding/faithfulness postulates, enumerated and drift-checked by `spec/check-trust-ledger.sh` - the build fails if this set changes without the ledger being updated.

Everything else is a definition or a machine-checked proof.

8.3 Non-stuckness (coverage)

```
fanout-empty-inhabited : ∀ {ctx cid hk n cp tfin ada} {crs}
  → tfin < ValidityInterval.lo (Context.validity ctx)
  → burnAllTokensOK ctx (emptyClosed cid hk n cp tfin ada)
  → fanoutValueOK ctx ada 0
  → crsBindOK ctx crs
  → ∃[ π ] FanoutValid ctx (emptyClosed cid hk n cp tfin ada) 0 π crs
```

```
finalize-reachable-empty : ∀ {ctx cid hk n cp tfin ada} {crs}
  → tfin < ValidityInterval.lo (Context.validity ctx)
  → burnAllTokensOK ctx (emptyClosed cid hk n cp tfin ada)
  → fanoutValueOK ctx ada 0
  → crsBindOK ctx crs
  → Reachable (emptyClosed cid hk n cp tfin ada)
    × (∃[ π ] FanoutValid ctx (emptyClosed cid hk n cp tfin ada) 0 π crs)
```

```
-- No `Reachable` premise: the membership witness comes from `accVerify-self`, so the bundle is
-- constructible from the contextual antecedents alone. Taking reachability and discarding it would
-- state something weaker than what is proved while reading as though reachability were needed -
-- contrast `progress-finalizable` below, which genuinely consumes its `Reachableᵛ` (via
-- `progress-nonEmpty`) to derive the final batch's `0 < m`.
fanout-coverage : ∀ {ctx cid hk n cp v s C tfin ada V crs}
  → distributedOuts ctx (setSize V) ≡ V
  → tfin < ValidityInterval.lo (Context.validity ctx)
  → burnAllTokensOK ctx (Closed cid hk n cp v s (accUTxO V) C tfin ada)
```

```
  → fanoutValueOK ctx ada (setSize V)
  → crsBindOK ctx crs
  → ∃[ π ] FanoutValid ctx (Closed cid hk n cp v s (accUTxO V) C tfin ada) (setSize V) π crs
```

```
progress-nonEmpty : ∀ {cid hk n tfin η ada}
  → Reachableᵛ (FanoutProgress cid hk n tfin η ada) → ¬ (η ≡ G₁)
```

```
progress-finalizable : ∀ {ctx cid hk n tfin ada V crs}
  → Reachableᵛ (FanoutProgress cid hk n tfin (accUTxO V) ada)
  → distributedOuts ctx (setSize V) ≡ V
  → tfin < ValidityInterval.lo (Context.validity ctx)
  → burnAllTokensOK ctx (FanoutProgress cid hk n tfin (accUTxO V) ada)
  → fanoutValueOK ctx ada (setSize V)
  → crsBindOK ctx crs
  → ∃[ π ] FinalPartialFanoutValid ctx (FanoutProgress cid hk n tfin (accUTxO V) ada) (setSize V) π crs
```

8.4 Safety invariants of reachable states

```
final-is-terminal : ∀ {r d'} → ¬ (Final ⁻⟨ r ⟩ d')
```

8.5 Value conservation (no theft)

```
fanout-conserves-ada : ∀ {ctx d m π crs}
  → (b : FanoutValid ctx d m π crs)
  → adaOf (headValueIn ctx)
    ≡ adaOf (takeSumᵛ m (Context.outputs ctx)) + (adaOf (burnedValue ctx) + adaOf (headAda d))
```

```
finalPartialFanout-conserves-ada : ∀ {ctx d m π crs}
  → (b : FinalPartialFanoutValid ctx d m π crs)
  → adaOf (headValueIn ctx)
    ≡ adaOf (takeSumᵛ m (Context.outputs ctx)) + (adaOf (burnedValue ctx) + adaOf (headAda d))
```

```
close-preserves-value : ∀ {ctx cid hk n cp v η ada s′ η′ C tfin ct}
  → (b : closeValid ctx (Open cid hk n cp v η ada) (Closed cid hk n cp v s′ η′ C tfin ada) ct)
  → (adaOf (headValueIn ctx) ≡ adaOf (headValue ctx)) × (nonAdaOf (headValueIn ctx) ≡ nonAdaOf (headValue
ctx))
```

```
contest-preserves-value : ∀ {ctx cid hk n cp v s η C tfin ada s′ η′ kh tfin′ ct}
  → (b : contestValid ctx (Closed cid hk n cp v s η C tfin ada) (Closed cid hk n cp v s′ η′ (kh :: C) tfin′
ada) ct)
  → (adaOf (headValueIn ctx) ≡ adaOf (headValue ctx)) × (nonAdaOf (headValueIn ctx) ≡ nonAdaOf (headValue
ctx))
```

```
partialFanout-conserves-ada : ∀ {ctx d d′ m crs}
  → (b : PartialFanoutValid ctx d d′ m crs)
  → adaOf (headValueIn ctx) ≡ adaOf (headValue ctx) + adaOf (decommitValue ctx m)
```

```
fanout-distributes-committed : ∀ {ctx cid hk n cp v s V C tfin ada m π crs}
  → FanoutValid ctx (Closed cid hk n cp v s (accUTxO V) C tfin ada) m π crs
  → distributedOuts ctx m ⊆ V
```

```
finalPartialFanout-distributes-committed : ∀ {ctx cid hk n tfin V ada m π crs}
  → FinalPartialFanoutValid ctx (FanoutProgress cid hk n tfin (accUTxO V) ada) m π crs
  → distributedOuts ctx m ⊆ V
```

```
init-reachable : ∀ {ctx seed cid hk n cp v η ada}
  → initValid ctx seed (Open cid hk n cp v η ada)
  → Reachable (Open cid hk n cp v η ada)
```

8.6 The contest game is bounded

```
deadline-bounded : ∀ {cid hk n cp v s η C tfin ada}
  → Reachableⱽ (Closed cid hk n cp v s η C tfin ada)
  → ∃[ hi ] (tfin ≤ (hi + cp) + length C * cp)
```

```
module ContestBound
  (ptKeysOf : ℍ → List ℍ)
  (pt-mem   : ∀ {ctx cid kh} → quantityOf (headValue ctx) (cid , kh) ≡ 1ℤ → kh ∈ˡ ptKeysOf cid)
  where

  contesters-are-participants : ∀ {cid hk n cp v s η C tfin ada}
    → Reachableⱽ (Closed cid hk n cp v s η C tfin ada)
    → ∀ {kh} → kh ∈ˡ C → kh ∈ˡ ptKeysOf cid
```

```
  contesters-bounded : ∀ {cid hk n cp v s η C tfin ada}
    → Reachableⱽ (Closed cid hk n cp v s η C tfin ada)
    → length C ≤ length (ptKeysOf cid)
```

```
  contest-window-bounded : ∀ {cid hk n cp v s η C tfin ada}
    → length (ptKeysOf cid) ≡ n
    → Reachableⱽ (Closed cid hk n cp v s η C tfin ada)
    → ∃[ hi ] (tfin ≤ (hi + cp) + n * cp)
```

8.7 The bridge to the extracted reference checker

```
close-spec⇒reference : ∀ ctx cid hk n cp v η ada s′ η′ C tfin
  → closeValid ctx (Open cid hk n cp v η ada) (Closed cid hk n cp v s′ η′ C tfin ada) closeInitial
  → (R.closeRefᵇ RB.mockOps (R.mkOpenᶜ v cp) (R.mkClosedᶜ v cp s′ (length C) tfin)
       (RB.closeTagOf closeInitial)
       (ValidityInterval.hi (Context.validity ctx)) (ValidityInterval.lo (Context.validity ctx)) ≡ true)
    × (R.valuePreservedᵇ (adaOf (headValueIn ctx)) (adaOf (headValue ctx))
                         (nonAdaOf (headValueIn ctx)) (nonAdaOf (headValue ctx)) ≡ true)
    × (R.noMintRefᵇ (RB.mintEntryCount ctx) ≡ true)
    × (R.participantSignedRefᵇ (R.mkSignerIOᶜ (RB.signerCodes ctx) (RB.ptCodes cid ctx)) ≡ true)
```

8.8 Machine-checked handler invariants

```
NoBothInFlight : LocalState → Set
NoBothInFlight st = (LocalState.currentDepositTxId st ≡ nothing) ⊎ (LocalState.pendingDecrement st ≡
nothing)
```

```
noBothInFlight-reachable : ∀ {init st} → NoBothInFlight init → Reachableᴴ init st → NoBothInFlight st
```

```
VersionDiscipline : LocalState → Set
VersionDiscipline st =
    (LocalState.seenVersion st ≡ Snapshot.version (LocalState.confirmed st))
  ⊎ (LocalState.seenVersion st ≡ suc (Snapshot.version (LocalState.confirmed st)))
```

```
versionDiscipline-reachable : ∀ {init st} → VersionDiscipline init → Reachableᴴ init st → VersionDiscipline
st
```

8.9 The signing message and unforgeability

```
-- MS-scheme unforgeability is a DERIVED THEOREM: a verifying aggregate ⇒
-- every party signed. It FACTORS through the scheme's decomposition (`aggSound`) and per-signature
-- unforgeability (`sigUnforge`) -- so the trusted base is the standard per-signature EUF-CMA assumption
-- plus the aggregation scheme's structure, not a monolithic aggregate-level axiom. Downstream uses
-- (`confirm`, `soundness`, `reflects`, `confCert-all`) consume it through this unchanged type.
ms-unforgeable : ∀ sys snap → AggVerified sys snap → Certified sys snap
```

8.10 Initial systems, reachability and the invariant

```
-- The DERIVED invariants carried through every reachable system, one per field. Each honest-signature
-- fact is DERIVED at the `signHonest` step from the `reqSn-sign` handler + the no-in-flight
-- precondition + the invariants (`signNumBound`/`sigSeen`):
--   sigApp   : every honest signature is on a snapshot applicable to U₀ (via `applyTxs-compose`);
--   sigDedup : an honest party signs at most one snapshot per number (via `signNumBound`);
--   confApp  : every honest party's confirmed snapshot is applicable to U₀ (L3), DERIVED rather than
--              assumed for the whole chain.
--   sigPos   : an honest signature is on a snapshot of number > 0 (the handler's s = s̄+1);
--   confCert : an honest party's confirmed snapshot is the genesis or is certified;
--   sigChain : every honest signature has an extending certified-or-genesis `PredecessorWitness`.
--              The last three give L2 (`confirmed-nest`).
record Inv (sys : System) : Set where
  field
    sigApp   : ∀ {k snap} → lookup (honest sys) k ≡ true → Signed sys k snap
             → Applicable (U₀ sys) (Snapshot.txs snap)
    sigDedup : ∀ {k s1 s2} → lookup (honest sys) k ≡ true → Signed sys k s1 → Signed sys k s2
             → Snapshot.number s1 ≡ Snapshot.number s2 → s1 ≡ s2
    confApp  : ∀ {i} → lookup (honest sys) i ≡ true
             → Applicable (U₀ sys) (confirmedTxs (lookup (localOf sys) i))
    sigPos   : ∀ {k snap} → lookup (honest sys) k ≡ true → Signed sys k snap → 0 < Snapshot.number snap
    confCert : ∀ {i} → lookup (honest sys) i ≡ true
             → (confirmedNo (lookup (localOf sys) i) ≡ 0 × confirmedTxs (lookup (localOf sys) i) ≡ [])
               ⊎ Certified sys (LocalState.confirmed (lookup (localOf sys) i))
    sigChain : ∀ {k snap} → lookup (honest sys) k ≡ true → Signed sys k snap → PredecessorWitness (Certified
sys) snap
    -- signNumBound: every honest signature's number is ≤ that party's last-signed number ŝ. With the
    --   `signHonest` no-in-flight precondition (ŝ = s̄) and the handler signing s = s̄+1 and bumping ŝ,
    --   this DERIVES the one-signature-per-round guard (`sigDedup`): a fresh sign is strictly above ŝ.
    signNumBound : ∀ {k snap} → lookup (honest sys) k ≡ true → Signed sys k snap
                 → Snapshot.number snap ≤ LocalState.seenNumber (lookup (localOf sys) k)
    -- sigSeen: every honest signature is on txs the party has SEEN. DERIVES the only-seen guard; from
    --   the handler's Δ ⊆ seen + (confirmedTxs ⊆ seen, itself from confCert+sigSeen). Feeds the
    --   second conjunct of `soundness`.
    sigSeen      : ∀ {k snap} → lookup (honest sys) k ≡ true → Signed sys k snap
                 → Snapshot.txs snap ⊆ˡ lookup (seen sys) k
```

8.11 The property statements

```
-- The §7 Consistency property: no two honest parties confirm conflicting transactions. We DERIVE
-- that each honest party's confirmed set is applicable to U₀ (`conf-applicable`) and that the two
-- sets nest (`confirmed-nest`); so their union is the larger set, which is applicable. "Conflicting"
-- means the union fails to apply, which nesting + individual applicability rules out. The union form
-- itself (a set T ⊇ both honest confirmed sets that is applicable to U₀) is machine-checked as
-- `consistency-union` (SecurityProofs), so the paper's `U₀ ∘ (T̄ᵢ ∪ T̄ⱼ) ≠ ⊥` is not left as prose.
HoldsAt : System → Set
HoldsAt sys =
  ∀ (i j : Fin (parties sys))
  → lookup (honest sys) i ≡ true → lookup (honest sys) j ≡ true
  → (confirmedTxs (lookup (localOf sys) i) ⊆ˡ confirmedTxs (lookup (localOf sys) j)
       ⊎ confirmedTxs (lookup (localOf sys) j) ⊆ˡ confirmedTxs (lookup (localOf sys) i))
  × Applicable (U₀ sys) (confirmedTxs (lookup (localOf sys) i))
  × Applicable (U₀ sys) (confirmedTxs (lookup (localOf sys) j))
```

```
Consistency : Set
Consistency = ∀ (sys : System) → Reachable sys → HoldsAt sys
```

```
-- ── Soundness and Completeness (Chain) ──────────────────────────────────────────
-- The finalized on-chain UTxO is the closed/fanned-out snapshot applied to U₀. That snapshot is
-- certified (the head closes only against a fully-signed snapshot), so by `cert-applicable` it is
-- conflict-free.
Ufinal : System → Snapshot → Maybe UTxO
Ufinal sys snap = applyTxs (U₀ sys) (Snapshot.txs snap)


-- Soundness (Chain), §7: the final UTxO U₀ ∘ T̃ for a finalized snapshot T̃ whose aggregate
-- multisignature verifies (`AggVerified`) is conflict-free AND its transactions were seen by EVERY
-- honest party (`T̃ ⊆ ∩_{j∈H} seen_j`). DERIVED: `ms-unforgeable` makes the verified snapshot certified
-- (every party signed it); each honest signer signed only applicable txs (`cert-applicable`, giving
-- conflict-freedom) it had seen (`sigSeen-inv`, giving the ∩-seen subset).
Soundness : Set
Soundness = ∀ sys → Reachable sys → ∀ {h snap} → lookup (honest sys) h ≡ true → AggVerified sys snap
          → Σ[ U ∈ UTxO ] (Ufinal sys snap ≡ just U)
                        × (∀ {j} → lookup (honest sys) j ≡ true → Snapshot.txs snap ⊆ˡ lookup (seen sys) j)
```

```
-- Completeness (Chain), §7: every transaction an honest party confirmed (T̄ᵢ) is included in the
-- FINALIZED snapshot T̃ (the closed/fanned-out snapshot, whose aggregate multisignature verifies),
-- for every honest party whose confirmed number is ≤ T̃'s. DERIVED: T̃ is certified
-- (`ms-unforgeable`), the honest party's confirmed snapshot is certified-or-genesis (`confCert-of`),
-- and two certified snapshots nest by number (`cert-nest`, L2). The `confirmedNo i ≤ number snap`
-- premise is the §7 "the finalized snapshot is at least as advanced as every honest confirmed one"
-- fact: in the real protocol the close/contest process always accepts the latest multi-signed
-- snapshot (so Ufinal.s ≥ maxᵢ s̄ᵢ); our `finalize` admits ANY certified snapshot, so it is a
-- per-party premise rather than derived. (The sibling honest-to-honest nesting is `confirmed-nest`.)
Completeness : Set
Completeness = ∀ sys → Reachable sys → ∀ {snap} → AggVerified sys snap
  → ∀ i → lookup (honest sys) i ≡ true
  → confirmedNo (lookup (localOf sys) i) ≤ Snapshot.number snap
  → confirmedTxs (lookup (localOf sys) i) ⊆ˡ Snapshot.txs snap
```

```
-- Bridge predicate: the on-chain head datum REFLECTS a finalized snapshot `snap` -- its snapshot
-- number matches and its stored accumulator commits (`OC.accUTxO`) to U₀ ∘ (txs snap).
record Reflects (sys : System) (snap : Snapshot) : Set where
  constructor mkReflects
  field
    finalUtxo     : UTxO
    conflictFree  : Ufinal sys snap ≡ just finalUtxo                 -- the final UTxO is conflict-free
    numberMatches : OC.snapNum (onChain sys) ≡ Snapshot.number snap   -- on-chain snapshot number matches
    accCommits    : OC.ηOf (onChain sys) ≡ OC.accUTxO (outsOf finalUtxo)  -- on-chain accumulator commits to
it
```

8.12 Machine-checked results

```
invariant : ∀ sys → Reachable sys → Inv sys
```

```
agree : ∀ sys → Reachable sys → ∀ {h s1 s2} → lookup (honest sys) h ≡ true
  → Certified sys s1 → Certified sys s2 → Snapshot.number s1 ≡ Snapshot.number s2 → s1 ≡ s2
```

```
cert-nest : ∀ sys → Reachable sys → ∀ {h s1 s2}
  → lookup (honest sys) h ≡ true → Certified sys s1 → Certified sys s2
  → Snapshot.number s1 ≤ Snapshot.number s2 → Snapshot.txs s1 ⊆ˡ Snapshot.txs s2
```

```
confirmed-nest : ∀ sys → Reachable sys → ∀ i j
  → lookup (honest sys) i ≡ true → lookup (honest sys) j ≡ true
  → confirmedNo (lookup (localOf sys) i) ≤ confirmedNo (lookup (localOf sys) j)
  → confirmedTxs (lookup (localOf sys) i) ⊆ˡ confirmedTxs (lookup (localOf sys) j)
```

8.13 Consistency

```
consistency : Consistency
```

```
consistency-uncorrupted : ∀ sys → Reachable sys → ∀ {h} → lookup (honest sys) h ≡ true → ∀ i j
  → (confirmedTxs (lookup (localOf sys) i) ⊆ˡ confirmedTxs (lookup (localOf sys) j)
       ⊎ confirmedTxs (lookup (localOf sys) j) ⊆ˡ confirmedTxs (lookup (localOf sys) i))
   × Applicable (U₀ sys) (confirmedTxs (lookup (localOf sys) i))
   × Applicable (U₀ sys) (confirmedTxs (lookup (localOf sys) j))
```

8.14 Soundness and completeness

```
soundness : Soundness
```

```
completeness : Completeness
```

8.15 The on-chain reflection bridge

```
reflects : ∀ sys → Reachable sys → ∀ {h snap}
  → lookup (honest sys) h ≡ true
  → AggVerified sys snap
  → OC.snapNum (onChain sys) ≡ Snapshot.number snap
  → (∀ U → Ufinal sys snap ≡ just U → OC.ηOf (onChain sys) ≡ OC.accUTxO (outsOf U))
  → Reflects sys snap
```

```
reflect-sound : ∀ sys → Reachable sys → ∀ {snap} → Reflects sys snap
  → Σ[ U ∈ UTxO ] (Ufinal sys snap ≡ just U)
                × (OC.ηOf (onChain sys) ≡ OC.accUTxO (outsOf U))
```

```
reflect-fanout-⊆ : ∀ sys → ∀ {ctx U m π crs}
  → OC.ηOf (onChain sys) ≡ OC.accUTxO (outsOf U)
  → OC.FanoutValid ctx (onChain sys) m π crs
  → OC.distributedOuts ctx m ⊆ outsOf U
```

8.16 No settlement without unanimity

```
sig-certifies : ∀ sys (snap : Snapshot) {hk cid v s η# δ# κ# ξ}
  → OC.snapshotSigOK hk cid v s η# δ# κ# ξ
  → hk ≡ aggKey
  → Snapshot.cid snap ≡ cid
  → Snapshot.version snap ≡ v
  → Snapshot.number snap ≡ s
  → Snapshot.etaHash snap ≡ η#
  → Snapshot.decHash snap ≡ δ#
  → Snapshot.comHash snap ≡ κ#
  → ξ ≡ aggSigOf sys snap
  → Certified sys snap
```

```
claimTx-certified : ∀ sys {ctx dd cid n cp v η ada headOut ξ s ref δ#} (snap : Snapshot)
  → OC.ClaimTxValid ctx dd (OC.Open cid aggKey n cp v η ada) headOut ξ s ref δ#
```

```
  → Snapshot.cid snap ≡ cid
  → Snapshot.version snap ≡ v
  → Snapshot.number snap ≡ s
  → Snapshot.etaHash snap ≡ hash (OC.ηOf headOut)
  → Snapshot.decHash snap ≡ δ#
  → Snapshot.comHash snap ≡ OC.depositCommitsHashOf ctx ref
  → ξ ≡ aggSigOf sys snap
  → Certified sys snap
    × (ValidityInterval.hi (Context.validity ctx) ≤ OC.DepositDatum.tRecover dd)
```

8.17 Handler invariants across adversarial executions

```
noBothInFlightˢ : ∀ sys → Reachable sys → ∀ i → NoBothInFlight (lookup (localOf sys) i)
```

```
versionDisciplineˢ : ∀ sys → Reachable sys → ∀ i → VersionDiscipline (lookup (localOf sys) i)
```

8.18 Solvency

```
data PendingShape (snap : Snapshot) : Set where
  -- a commit is pending: κ# binds the recorded commit set's hash and the deposit's
  -- transaction id (node `commitOutputsHash`); `incVal` is the recorded set's value
  -- (`observeDepositTx` refuses deposits whose value differs); no decommit digest.
  pendingCommit :
    (cHash depId : ℍ) (incVal : Value)
    → Snapshot.comHash snap ≡ hash (cHash ‖ depId)
    → depId ≢ noTxId
    → Snapshot.decHash snap ≡ hash {A = List Output} []
    → PendingShape snap
  -- a decommit is pending: δ# binds the decommitted output list; no commit digest.
  pendingDecommit :
    (decOuts : List Output)
    → Snapshot.decHash snap ≡ hash decOuts
    → Snapshot.comHash snap ≡ hash (noCommitHash ‖ noTxId)
    → PendingShape snap
  -- neither is pending.
  pendingNone :
    Snapshot.comHash snap ≡ hash (noCommitHash ‖ noTxId)
    → Snapshot.decHash snap ≡ hash {A = List Output} []
    → PendingShape snap

record HonestFacts (snap : Snapshot) : Set where
  field
    -- the L2 UTxO set the snapshot's accumulator commits to (the honest node computes
    -- η from the snapshot UTxO it signs).
    committed : ℙ Output
    ηCoheres  : Snapshot.etaHash snap ≡ hash (OC.accUTxO committed)
    shape     : PendingShape snap
```

```
record Assumptions (sys : System) : Set₁ where
  field
    -- the commit digest κ# = hash(commit-list hash ‖ deposit tx id) determines the
    -- deposit's transaction id. This hypothesis is only applicable because
    -- `depositCommitsHashOf` binds the tx id by definition: under the pre-fix
    -- digest hash(C) it says nothing, and the increment case below cannot close.
    κ#-pair-inj : ∀ {x y r s : ℍ} → hash (x ‖ r) ≡ hash (y ‖ s) → r ≡ s
    -- the accumulator hash η# determines the commitment.
    η#-inj      : ∀ {a b : AccCommitment} → hash a ≡ hash b → a ≡ b
    -- the decommit digest δ# determines the output list.
    outs#-inj   : ∀ {xs ys : List Output} → hash xs ≡ hash ys → xs ≡ ys
    -- a certificate implies the honest-signing facts above.
    honest-certified : ∀ {snap : Snapshot} → Certified sys snap → HonestFacts snap
```

```
-- Guard for reviewers: this hypothesis is stated against the _observed_ deposit id
-- (`depId`, what the parties signed for), never against the claimed ref. That
-- placement is the load-bearing modeling choice of the whole invariant - restated
-- at the claimed ref it would make solvency vacuously provable for the pre-fix
-- protocol, which the counter-model (`SolvencyCounterModel`) shows is unsound.
-- The proof must earn the crossing from claimed ref to observed deposit, and can
-- only do so through the transaction-id binding in the signed digest.
ObservedDeposit : Context → ∀ {snap} → PendingShape snap → Set
ObservedDeposit ctx (pendingCommit _ depId incVal _ _ _) =
  ∀ (r : OutputRef) → OutputRef.txId r ≡ depId → OutputRef.index r ≡ 0
    → OC.depositValueAt ctx r ≡ incVal
ObservedDeposit _ _ = ⊤

IncCoherent : ℙ Output → ℙ Output → ∀ {snap} → PendingShape snap → Set
IncCoherent U U' (pendingCommit _ _ incVal _ _ _) = sumValue U' ≡ sumValue U +ᵛ incVal
IncCoherent _ _ _ = ⊤

DecCoherent : ℙ Output → ℙ Output → ∀ {snap} → PendingShape snap → Set
DecCoherent U U' (pendingDecommit decOuts _ _) = sumValue U ≡ sumValue U' +ᵛ listValue decOuts
DecCoherent _ _ _ = ⊤
```

```
data SolventReach (r₀ : Value) : OC.HeadDatum → ℙ Output → Value → Set where
  s-init : ∀ {ctx seed cid n cp v η ada}
    → OC.InitValid ctx seed cid n v η
    → OC.headValue ctx ≡ r₀
    → SolventReach r₀ (OC.Open cid aggKey n cp v η ada) ∅ˢ r₀

  s-inc : ∀ {ctx cid n cp v η η' ada ξ s ref δ# U w} {snap : Snapshot}
    → SolventReach r₀ (OC.Open cid aggKey n cp v η ada) U w
    → (b : OC.IncrementValid ctx aggKey cid v (OC.Open cid aggKey n cp v η ada)
                             (OC.Open cid aggKey n cp (suc v) η' ada) ξ s ref δ#)
    → (hf : HonestFacts snap)
    → Snapshot.etaHash snap ≡ hash η'                          -- unforgeability: η# is the signed one
    → Snapshot.comHash snap ≡ OC.depositCommitsHashOf ctx ref  -- unforgeability: κ# is the signed one
    → OC.headValueIn ctx ≡ w                                   -- L1 continuity (ledger)
    → OutputRef.txId ref ≢ noTxId                              -- a spent out-ref's tx id is real (ledger)
    → ObservedDeposit ctx (HonestFacts.shape hf)
    → IncCoherent U (HonestFacts.committed hf) (HonestFacts.shape hf)
    → SolventReach r₀ (OC.Open cid aggKey n cp (suc v) η' ada)
                  (HonestFacts.committed hf) (OC.headValue ctx)

  s-dec : ∀ {ctx cid n cp v η η' ada ξ s m κ# U w} {snap : Snapshot}
    → SolventReach r₀ (OC.Open cid aggKey n cp v η ada) U w
    → (b : OC.DecrementValid ctx aggKey cid v (OC.Open cid aggKey n cp v η ada)
                             (OC.Open cid aggKey n cp (suc v) η' ada) ξ s m κ#)
    → (hf : HonestFacts snap)
    → Snapshot.etaHash snap ≡ hash η'                          -- unforgeability: η# is the signed one
    → Snapshot.decHash snap ≡ OC.decommitOutputsHashOf ctx m   -- unforgeability: δ# is the signed one
    → OC.headValueIn ctx ≡ w                                   -- L1 continuity (ledger)
    → DecCoherent U (HonestFacts.committed hf) (HonestFacts.shape hf)
    → SolventReach r₀ (OC.Open cid aggKey n cp (suc v) η' ada)
                  (HonestFacts.committed hf) (OC.headValue ctx)

  -- closing on the initial (empty) snapshot: the stored accumulator is the empty
  -- commitment and the head value is preserved exactly.
  s-closeInitial : ∀ {ctx cid n cp v η ada s' η' C tfin U w}
    → SolventReach r₀ (OC.Open cid aggKey n cp v η ada) U w
    → OC.CloseValid ctx aggKey cid v cp s' (OC.Open cid aggKey n cp v η ada)
                    (OC.Closed cid aggKey n cp v s' η' C tfin ada) OC.closeInitial
    → OC.headValueIn ctx ≡ w                                   -- L1 continuity (ledger)
    → SolventReach r₀ (OC.Closed cid aggKey n cp v s' η' C tfin ada) ∅ˢ (OC.headValue ctx)

  -- closing on a certified snapshot at the current version (the closeUnused
  -- redeemer; closeAny differs only in its snapshot-number conjunct and follows
  -- identically). The stored accumulator jumps to the closing snapshot's s; the
```

```
  -- head value is preserved exactly, so the per-step hypothesis is that the
  -- closing snapshot's committed value equals the settled one - L2 transactions
  -- between settlements preserve value (owner: the L2 ledger rules).
  s-close : ∀ {ctx cid n cp v η ada ξ η# δ# κ# s' η' C tfin U w} {snap : Snapshot}
    → SolventReach r₀ (OC.Open cid aggKey n cp v η ada) U w
    → OC.CloseValid ctx aggKey cid v cp s' (OC.Open cid aggKey n cp v η ada)
                    (OC.Closed cid aggKey n cp v s' η' C tfin ada) (OC.closeUnused ξ η# δ# κ#)
    → (hf : HonestFacts snap)
    → Snapshot.etaHash snap ≡ η#                                -- unforgeability: η# is the signed one
    → OC.headValueIn ctx ≡ w                                    -- L1 continuity (ledger)
    → sumValue (HonestFacts.committed hf) ≡ sumValue U          -- L2 value preservation
    → SolventReach r₀ (OC.Closed cid aggKey n cp v s' η' C tfin ada)
                  (HonestFacts.committed hf) (OC.headValue ctx)

  -- contesting with a newer certified snapshot (the contestUnused redeemer;
  -- contestUsed combines a pending delta into the stored accumulator and is
  -- future work, as is closeUsed). Same jump-and-preserve pattern as close.
  s-contest : ∀ {ctx cid n cp v s η C tfin ada ξ η# δ# κ# s' η' kh tfin' U w} {snap : Snapshot}
    → SolventReach r₀ (OC.Closed cid aggKey n cp v s η C tfin ada) U w
    → OC.ContestValid ctx aggKey cid v s tfin (OC.Closed cid aggKey n cp v s η C tfin ada)
                      (OC.Closed cid aggKey n cp v s' η' (kh :: C) tfin' ada)
                      (OC.contestUnused ξ η# δ# κ#) kh
    → (hf : HonestFacts snap)
    → Snapshot.etaHash snap ≡ η#                                -- unforgeability: η# is the signed one
    → OC.headValueIn ctx ≡ w                                    -- L1 continuity (ledger)
    → sumValue (HonestFacts.committed hf) ≡ sumValue U          -- L2 value preservation
    → SolventReach r₀ (OC.Closed cid aggKey n cp v s' η' (kh :: C) tfin' ada)
                  (HonestFacts.committed hf) (OC.headValue ctx)
```

```
module Invariant {sys : System} (H : Assumptions sys) where
  open Assumptions H

  solvency : ∀ {r₀ d U w}
    → SolventReach r₀ d U w
    → (OC.ηOf d ≡ OC.accUTxO U) × (w ≡ r₀ +ᵛ sumValue U)
```

```
  s-inc-certified : ∀ {r₀ ctx cid n cp v η η' ada ξ s ref δ# U w} {snap : Snapshot}
    → SolventReach r₀ (OC.Open cid aggKey n cp v η ada) U w
    → (b : OC.IncrementValid ctx aggKey cid v (OC.Open cid aggKey n cp v η ada)
                             (OC.Open cid aggKey n cp (suc v) η' ada) ξ s ref δ#)
    → (cert : Certified sys snap)
    → Snapshot.etaHash snap ≡ hash η'
    → Snapshot.comHash snap ≡ OC.depositCommitsHashOf ctx ref
    → OC.headValueIn ctx ≡ w
    → OutputRef.txId ref ≢ noTxId
    → ObservedDeposit ctx (HonestFacts.shape (honest-certified cert))
    → IncCoherent U (HonestFacts.committed (honest-certified cert))
                    (HonestFacts.shape (honest-certified cert))
    → SolventReach r₀ (OC.Open cid aggKey n cp (suc v) η' ada)
                  (HonestFacts.committed (honest-certified cert)) (OC.headValue ctx)

  s-dec-certified : ∀ {r₀ ctx cid n cp v η η' ada ξ s m κ# U w} {snap : Snapshot}
    → SolventReach r₀ (OC.Open cid aggKey n cp v η ada) U w
    → (b : OC.DecrementValid ctx aggKey cid v (OC.Open cid aggKey n cp v η ada)
                             (OC.Open cid aggKey n cp (suc v) η' ada) ξ s m κ#)
    → (cert : Certified sys snap)
    → Snapshot.etaHash snap ≡ hash η'
    → Snapshot.decHash snap ≡ OC.decommitOutputsHashOf ctx m
    → OC.headValueIn ctx ≡ w
    → DecCoherent U (HonestFacts.committed (honest-certified cert))
                    (HonestFacts.shape (honest-certified cert))
    → SolventReach r₀ (OC.Open cid aggKey n cp (suc v) η' ada)
                  (HonestFacts.committed (honest-certified cert)) (OC.headValue ctx)
```

```
fanout-covered : ∀ {r₀ d U w ctx m π crs}
  → SolventReach r₀ d U w
  → (fb : OC.FanoutValid ctx d m π crs)
  → OC.headValueIn ctx ≡ w                                    -- L1 continuity (ledger)
  → (OC.distributedOuts ctx m ⊆ U)
    × (w ≡ (OC.takeSumᵛ m (Context.outputs ctx) +ᵛ (OC.burnedValue ctx +ᵛ OC.headAda d)))
```

Bibliography

1. Chakravarty MM, Coretti S, Fitzi M, et al (2020) Hydra: Fast isomorphic state channels. Cryptology ePrint Archive
2. Hydra repository. https://github.com/cardano-scaling/hydra
3. Chakravarty MMT, Chapman J, MacKenzie K, et al (2020) The Extended UTxO Model. In: 4th Workshop on Trusted Smart Contracts
4. Extended UTXO-2 model. https://github.com/hydra-supplementary-material/eutxo-spec/blob/master/extended-utxo-specification.pdf
5. Itakura K, Nakamura K (1983) A public-key cryptosystem suitable for digital multisignatures. NEC Research & Development 1–8
6. Micali S, Ohta K, Reyzin L (2001) Accountable-Subgroup Multisignatures: Extended Abstract. In: Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS 2001). ACM, pp 245–254
7. Atzei N, Bartoletti M, Lande S, Zunino R (2018) A Formal Model of Bitcoin Transactions. In: Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers. pp 541–560
8. Zahnentferner J (2018) An Abstract Model of UTxO-based Cryptocurrencies with Scripts. IACR Cryptology ePrint Archive 2018:469
9. Chakravarty MMT, Chapman J, Mackenzie KM, et al (2020) UTXOma: UTXO with Multi-Asset Support
10. A Formal Specification of the Cardano Ledger integrating Plutus Core. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/alonzo-ledger.pdf
11. A Formal Specification of the Cardano Ledger. https://github.com/input-output-hk/cardano-ledger/releases/latest/download/shelley-ledger.pdf
12. Kate A, Zaverucha GM, Goldberg I (2010) Constant-Size Commitments to Polynomials and Their Applications. In: Advances in Cryptology – ASIACRYPT 2010. pp 177–194