Anoma Research Topics | TECHNICAL REPORT

Anoma Resource Machine Specification

Yulia Khalniyazova[a] and Christopher Goes[a]

[a]Heliax AG

* *E-Mail: {yulia, cwgoes}@heliax.dev*

Abstract

The article explores the Anoma Resource Machine (ARM) within the Anoma protocol, providing a comprehensive understanding of its role in facilitating state updates based on user preferences. Drawing parallels with the Ethereum Virtual Machine, the ARM introduces a flexible transaction model, diverging from traditional account and UTXO models. Key properties such as atomic state transitions, information flow control, account abstraction, and an intent-centric architecture contribute to the ARM's robustness and versatility. Inspired by the Zcash protocol, the ARM leverages commitment accumulators to ensure transaction privacy. The article outlines essential building blocks, computable components, and requirements for constructing the ARM, highlighting its unique approach to resource-based state management.

Keywords: Anoma Resource Machine ; resource model ; virtual machine ; transaction privacy ;

(Received Nov 10, 2023; Revised Jan 25, 2024 ; Version: Jan 25, 2024)

Contents

DOI: 10.5281/zenodo.10498991

1. Introduction

In the Anoma protocol, users submit preferences about the system state and the system continuously updates its state based on those preferences. The Anoma Resource Machine (ARM) is the part of the Anoma protocol that defines and enforces the rules for valid state updates that satisfy users' preferences. The new proposed state is then agreed on by the consensus participants. In that sense the role of the Anoma Resource Machine in the Anoma protocol is similar to the role of the Ethereum Virtual Machine in the Ethereum protocol.

The atomic unit of the ARM state is called a **resource**. Resources are immutable, they can be created once and consumed once, which indicates the system state change. The resources that were created but not consumed yet make the current state of the system.

The ARM transaction model is neither the account nor UTXO model. Unlike the Bitcoin UTXO model, which sees UTXOs as currency units and is limited in expressivity, the resource model is generalised and provides flexibility — resource logics can be defined in a way to construct applications that operate in either transaction model (or both). For example, a token operating in the account model would be represented by a single resource containing a map $user : balance$ (unlike the UTXO model, where the token would be represented by a collection of resources of the token type, each of which would correspond to a portion of the token total supply and belong to some user owning this portion). Only one resource of that kind can exist at a time. When a user wants to perform a transfer, they consume the old balance table resource and produce a new balance table resource.

The Anoma Resource Machine has the following properties:

- *Atomic state transitions of unspecified complexity* — the number of resources created and consumed in every atomic state transition is not limited by the system.
- *Information flow control* — the parties involved in a transaction can decide how much of the information about their state to reveal and to whom. From the resource machine perspective, these states are treated equally (e.g., there is no difference between transparent and shielded resources), but the amount of information revealed about the states differs. It is realised with the help of *shielded execution*, in which the state transition is only visible to the parties involved.

- *Account abstraction* — each resource is controlled by a *resource logic* — a custom predicate that encodes constraints on valid state transitions for that kind of resource and determines if a resource can be created or consumed. A valid state transition requires a resource logic proof for every resource created or consumed in the proposed state transition.

- *Intent-centric architecture* — the ARM provides means to express intents and ensures their correct and complete fulfilment and settlement.

The design of the Anoma Resource Machine was significantly inspired by the Zcash protocol (Hopwood et al. (2023)).

The rest of the document contains the definitions of the ARM building blocks and the necessary and sufficient requirements to build the Anoma Resource Machine.

2. Preliminaries

2.1. Notation. For a function h, we denote the output finite field of h as $\mathbb{F}_h$. If a function h is used to derive a component x, we refer to the function as h_x, and the corresponding to h finite field is denoted as $\mathbb{F}_{h_x}$, or, for simplicity, $\mathbb{F}_x$.

3. Resource

A *resource* is a composite structure $R = (l, label, q, v, eph, nonce, npk, rseed) : Resource$ where:

- $Resource = \mathbb{F}_l \times \mathbb{F}_{label} \times \mathbb{F}_Q \times \mathbb{F}_v \times \mathbb{F}_b \times \mathbb{F}_{nonce} \times \mathbb{F}_{npk} \times \mathbb{F}_{rseed}$

- $l : \mathbb{F}_l$ is a succinct representation of the predicate associated with the resource (resource logic)

- $label : \mathbb{F}_{label}$ specifies the fungibility domain for the resource. Resources within the same fungibility domain are seen as equivalent kinds of different quantities. Resources from different fungibility domains are seen and treated as distinct asset kinds. This distinction comes into play in the balance check described later.

- $q : \mathbb{F}_Q$ is an number representing the quantity of the resource

- $v : \mathbb{F}_v$ is the fungible data of the resource. It contains extra information but does not affect the resource's fungibility

- $eph : \mathbb{F}_b$ is a flag that reflects the resource's ephemerality. Ephemeral resources do not get checked for existence when being consumed

- $nonce : \mathbb{F}_{nonce}$ guarantees the uniqueness of the resource computable components

- $npk : \mathbb{F}_{npk}$ is a nullifier public key. Corresponds to the nullifier key nk used to derive the resource nullifier (nullifiers are further described in 3.1.2)

- $rseed : \mathbb{F}_{rseed}$: randomness seed used to derive whatever randomness needed

To distinguish between the resource data structure consisting of the resource components and a resource as a unit of state identified by just one (or some) of the resource computed fields, we sometimes refer to the former as a resource plaintext.

3.1. Computable Components. Resource computable components are the components that are derivable from the resource components, other computed components, and possibly some secret data by applying a function from class H.

3.1.1. Resource Commitment.

Information flow control property implies working with flexible privacy requirements, varying from transparent contexts, where almost everything is publicly known, to contexts with stronger privacy guarantees, where as little information as possible is revealed.

From the resource model perspective, stronger privacy guarantees require operating on resources that are not publicly known in a publicly verifiable way. Therefore, proving the resource's existence has to be done without revealing the resource's plaintext.

One way to achieve this would be to publish a **commitment** to the resource plaintext. For a resource r, the resource commitment is computed as $r.cm = h_{cm}(r)$. Resource commitment has binding and hiding properties, meaning that the commitment is tied to the created resource but does not reveal information about the resource beyond the fact of creation. From the moment the resource is created, and until the moment it is consumed, the resource is a part of the system's state

Remark 1. The resource commitment is also used as the resource's address $r.addr$ in the content-addressed storage.

All resource commitments are stored in an append-only data structure called a **commitment accumulator** $CMacc$. Every time a resource is created, its commitment is added to the commitment accumulator. The resource commitment accumulator $CMacc$ is external to the resource machine, but the resource machine can read from it. A commitment accumulator is a cryptographic accumulator (Özçelik et al. (2021)) that allows to prove membership for elements accumulated in it, provided a witness and the accumulated value.

Each time a commitment is added to the $CMacc$, the accumulator and all witnesses of the already accumulated commitments are updated. For a commitment that existed in the accumulator before a new one was added, both the old witness and the new witness (with the corresponding accumulated value parameter) can be used to prove membership. However, the older the witness (and, consequently, the accumulator) that is used in the proof, the more information about the resource it reveals (the resource had to exist before it was consumed, and an older accumulator gives more concrete boundaries on the resource's creation time). For that reason, it is recommended to use fresher parameters when proving membership.

The commitment accumulator must support the following functionality:

- $WRITE(cm)$ adds an element to the accumulator, returning the witness used to prove membership.
- $WITNESS(cm)$ for a given element, returns the witness used to prove membership if the element is present, otherwise returns nothing.
- $VERIFY(cm, w, acc)$ verifies the membership proof for an element cm with a membership witness w in the accumulator acc.
- $ACC()$ returns the accumulator.

Currently, the commitment accumulator is assumed to be a Merkle tree $CMtree$ of depth $depth_{CMtree}$, where the leaves contain the resource commitments and the intermediate nodes' values are computed using a hash function h_{CMtree}.

Remark 2. The hash function h_{CMtree} used to compute the nodes of the $CMtree$ Merkle tree is not necessarily the same as the function used to compute commitments stored in the tree h_{cm}.

For a Merkle tree, the witness is the path to the resource commitment, and the tree root represents the accumulated value. To support the systems with stronger privacy requirements, the witness for such a proof must be a private input (4) when proving membership.

3.1.2. Resource Nullifier. A resource nullifier is a computed field, the publishing of which consumes an existing resource. For a resource r, the nullifier is computed from the resource's plaintext and a key called a nullifier key: $r.nf = h_{nf}(nk, r)$. A resource can be consumed only once. Nullifiers of consumed resources are stored in a public add-only structure called the resource nullifier set ($NFset$). This structure is external to the resource machine, but the resource machine can read from it.

Every time a resource is consumed, it has to be checked that the resource existed before (the resource's commitment is in the $CMtree$) and has not been consumed yet (the resource's nullifier is not in the $NFset$).

The nullifier set must support the following functionality:

- $WRITE(nf)$ adds an element to the nullifier set.
- $EXISTS(nf)$ checks if the element is present in the set, returning a boolean

3.1.3. Resource Kind. For a resource r, its kind is computed as $r.kind = h_{kind}(r.l, r.label)$.

3.1.4. Resource Delta. Resource deltas are used to reason about the total quantities of different kinds of resources in transactions. For a resource r, its delta is computed as $r.\Delta = h_\Delta(r.kind, r.q)$.

Remark 3. The function used to derive $r.\Delta$ must have the following properties:

- For resources of the same kind $kind$, h_Δ should be *additively homomorphic*: $r_1.\Delta + r_2.\Delta = h_\Delta(kind, r_1.q + r_2.q)$
- For resources of different kinds, h_Δ has to be *kind-distinct*: if there exists $kind$ and q s.t. $h_\Delta(r_1.kind, r_1.q) + h_\Delta(r_2.kind, r_2.q) = h_\Delta(kind, q)$, it is computationally infeasible to compute $kind$ and q.

An example of a function that satisfies these properties is the Pedersen commitment scheme: it is additively homomorphic, and its kind-distinctness property comes from the discrete logarithm assumption.

4. Proving system

A proving system allows proving statements about resources. Depending on the security requirements, a proving system might be instantiated, for example, by a signature scheme, a zk-SNARK, or a trivial transparent system where the properties are proven by openly verifying the properties of published data.

To support the intended spectrum of privacy requirements, varying from the strongest (where the relationship between the published parameters does not allow an observer to infer any kind of meaningful information about the state transition) to the weakest, where no privacy is required, we divide the proving system inputs into public (instance) and private (witness). The inputs that could potentially reveal the connection between components or other kinds of sensitive information are usually considered private, and the components that have to be and can be safely published regardless of the privacy guarantees of the system would be public inputs. Note that in the context of a fully public system, this distinction is not meaningful because all inputs are public in such a system.

We define a set of structures required to define a proving system PS as follows:

- Proof $\pi : PS.Proof$
- Instance $x : PS.Instance$ is the public input used to produce a proof.
- Witness $w : PS.Witness$ is the private input used to produce a proof.

- Proving key $pk : PS.ProvingKey$ contains the secret data required to produce a proof for a pair (x, w).
- Verifying key $vk : PS.VerifyingKey$ contains the data required, along with the witness x, to verify a proof π.

A **proof record** carries the components required to verify a proof. It is defined as a composite structure $PR = (\pi, x, vk) : ProofRecord$, where:

- $ProofRecord = PS.VerifyingKey \times PS.Instance \times PS.Proof$
- $vk : PS.VerifyingKey$
- $x : PS.Instance$
- $\pi : PS.Proof$ is the proof of the desired statement.

A proving system PS consists of a pair of algorithms, $(Prove, Verify)$:

- $Prove(pk, x, w) : PS.ProvingKey \times PS.Instance \times PS.Witness \to PS.Proof$
- $Verify(pr) : PS.ProofRecord \to \mathbb{F}_b$

A proving system used to produce ARM proofs should have the following properties (as defined in Thaler (2023)):

- *Completeness*. This property states that any true statement should have a convincing proof of its validity.
- *Soundness*. This property states that no false statement should have a convincing proof.
- Proving systems used to provide privacy should additionally be *zero-knowledge*, meaning that the produced proofs reveal no information other than their own validity.

A proof π for which $Verify(pr) = 1$ is considered valid.

The party responsible for creating proofs is also responsible for providing the input to the proving system. Public inputs are required to verify the proof and must be available to any party that verifies the proof; private inputs do not have to be available and can be stored locally by the proof creator. The same rule applies to custom (inputs not specified by the ARM) public and private inputs.

5. Roles and requirements

The table below contains a list of resource-related roles. In the Anoma protocol, the role of the resource creator will often be taken by a solver, which creates additional security requirements compared to the case when protocol users solve their own intents. Because of that, extra measures are required to ensure reliable distribution of the information about the created resource to the resource receiver.

Role	**Description**
Authorizer	approves the resource consumption on the application level. The resource logic encodes the mechanism that connects the authorizer's external identity (public key) to the decision-making process
Annuler	knows the data required to nullify a resource
Creator	creates the resource and shares the data with the receiver
Owner	can both authorize and annul a resource
Sender	owns the resources that were consumed to create the created resource
Receiver	owns the created resource

Table 1: Resource-related roles.

DOI: 10.5281/zenodo.10498991

5.1. Reliable resource plaintext distribution. In the case of in-band distribution of created resources in contexts with higher security requirements, the resource creator is responsible for encrypting the resource plaintext. Verifiable encryption must be used to ensure the correctness of the encrypted data: the encrypted plaintext must be proven to correspond to the resource plaintext, passed as a private input.

5.2. Reliable nullifier key distribution. Knowing the resource's nullifier reveals information about when the resource is consumed, as the nullifier will be published when it happens. For that reason, it is advised to keep the number of parties who can compute the resource's nullifier as low as possible.

In particular, the resource creator should not be able to compute the resource nullifier, and as the nullifier key allows to compute the resource's nullifier, it shouldn't be known to the resource creator. At the same time, the resource plaintext must contain some information about the nullifier key. One way to ensure both requirements is, instead of sharing the nullifier key itself with the resource creator, to share some parameter derived from the nullifier key, but that does not allow computing the nullifier key or any meaningful information about it. This parameter is called a nullifier public key and is computed as $npk = h_{npk}(nk)$.

6. Transaction

A transaction is a composite structure $TX = (rts, cms, nfs, \Pi, \Delta, extra, \Phi)$, where:

- $rts \subseteq \mathbb{F}_{rt}$ is a set of roots of $CMtree$
- $cms \subseteq \mathbb{F}_{cm}$ is a set of created resources' commitments.
- $nfs \subseteq \mathbb{F}_{nf}$ is a set of consumed resources' nullifiers.
- $\Pi : \{\pi : ProofRecord\}$ is a set of proof records.
- $\Delta_{tx} : \mathbb{F}_\Delta$ is computed from Δ parameters of created and consumed resources. It represents the total delta change induced by the transaction.
- $extra : \{(k, d) : k \in \mathbb{F}_{key}, d \subseteq \mathbb{F}_d\}$ contains extra information requested by the logics of created and consumed resources.
- $\Phi : PREF$ where $PREF = TX \rightarrow [0, 1]$ is a preference function that takes a transaction as input and outputs a normalised value in the interval $[0, 1]$ that reflects the users' satisfaction with the produced transaction. For example, a user who wants to receive at least $q = 5$ of resource of kind A for a fixed amount of resource of kind B might set the preference function to implement a linear function that returns 0 at $q = 5$ and returns 1 at $q = q_{max} = |\mathbb{F}_q| - 1$.

6.1. Transaction balance change. Δ_{tx} of a transaction is computed from the delta parameters of the resources (3.1.4) consumed and created in the transaction. It represents the total quantity change per resource kind induced by the transaction. From the homomorphic properties of h_Δ, for the resources of the same kind $kind$: $\sum_j h_\Delta(kind, r_{i_j}.q) - \sum_j h_\Delta(kind, r_{o_j}.q) = \sum_j r_{i_j}.\Delta - \sum_j r_{o_j}.\Delta = h_\Delta(kind, q_{kind})$. The kind-distinctness property of h_Δ allows to compute $\Delta_{tx} = \sum_j r_{i_j}.\Delta - \sum_j r_{o_j}.\Delta$ adding resources of all kinds together without the need to explicitly distinguish between the resource kinds: $\sum_j r_{i_j}.\Delta - \sum_j r_{o_j}.\Delta = \sum_j h_\Delta(kind_j, q_{kind_j})$

6.2. Proofs. Each transaction refers to a set of resources to be consumed and a set of resources to be created. Creation and consumption of a resource requires a set of proofs that attest to the correctness of the underlying state transition. There are three proof types associated with a transaction:

- Resource logic proof π_{RL}. For each resource consumed or created in a transaction, it is required to provide a proof that the logic of the resource evaluates to 1 given the input parameters that describe the state transition (the resource machine instantiation defines the exact parameters).
- A delta proof (balance proof) π_Δ makes sure that Δ_{tx} is correctly derived from Δ parameters of the resources created and consumed in the transaction and commits to the expected publicly known value, called a *balancing value*.
- A resource machine compliance proof π_{compl} is required to ensure that the provided transaction is well-formed. The resource machine compliance proof must check that each consumed resource was consumed strictly after it was created, that the resource commitments and nullifiers are derived according to the commitment and nullifier derivation rules, and that the resource logics of created and consumed resources are satisfied.

Remark 4. It must also be checked that the created resource was created exactly once and the consumed resource was consumed exactly once. These checks can be performed separately, with read access to the $CMtree$ and $NFset$.

Remark 5. Every proof is created with a proving system PS and has the type $PS.Proof$. The proving system might differ for different proof types.

Remark 6. For privacy-preserving contexts, all proving systems in use should support data privacy, and the proving system used to create resource logic proofs should provide function privacy in addition to data privacy: provided proofs of two different circuits, an observer should not be able to tell which proof corresponds to which circuit. It is a stronger requirement than data privacy, which implies that an observer does not know the private input used to produce the proof.

6.3. Composition. Having two transactions tx_1 and tx_2, their composition $tx_1 \circ tx_2$ is defined as a transaction tx, where:

- $rts_{tx} = rts_1 \cup rts_2$
- $cms_{tx} = cms_1 \sqcup cms_2$
- $nfs_{tx} = nfs_1 \sqcup nfs_2$
- Proofs:
  - delta proof: $\Pi^{\Delta}_{tx} = AGG(\Pi^{\Delta}_1, \Pi^{\Delta}_2)$, where AGG is an aggregation function s.t. for bv_1 being the balancing value of the first delta proof, bv_2 being the balancing value of the second delta proof, and bv_{tx} being the balancing value of the composed delta proof, it satisfies $bv_{tx} = bv_1 + bv_2$. The aggregation function takes two delta proofs as input and outputs a delta proof.
  - resource logic proofs: $\Pi^{RL}_{tx} = \Pi^{RL}_1 \sqcup \Pi^{RL}_2$
  - compliance proofs: $\Pi^{compl}_{tx} = \Pi^{compl}_1 \sqcup \Pi^{compl}_2$
- $\Delta_{tx} = \Delta_1 + \Delta_2$
- $extra_{tx} = extra_1 \cup extra_2$
- $\Phi_{tx} = G(\Phi_1, \Phi_2)$, where $G : PREF \times PREF \rightarrow PREF$, and G is a preference function composition function

Remark 7. Composing sets with disjoint union operator $\sqcup$, it has to be checked that those sets do not have any elements in common. Otherwise, the transactions are not composable.

6.4. Validity. A transaction is considered *valid* if the following statements hold:

- rts contains valid $CMtree$ roots that are correct inputs for the membership proofs
- input resources have valid resource logic proofs and the compliance proofs associated with them
- output resources have valid resource logic proofs and the compliance proofs associated with them
- Δ is computed correctly and its opening is equal to the balancing value for that transaction

7. Resource Machine

A resource machine is a deterministic stateless machine that creates, composes, and verifies transaction candidates.

It has read-only access to the external global state, which includes the content-addressed storage system (which in particular stores resources), global commitment accumulator, and the global nullifier set, and can produce writes to the external local state that will later be applied to the system state by another entity.

The resource machine has two layers: the outer layer, the resource machine shell, that creates and processes *transaction candidates*, and the inner layer, the resource machine core, that creates and processes *transactions*. We assume the shell is simple in this version: it only validates the transaction candidate without any verification steps. The result is a transaction that is then passed to the core. The distribution of responsibilities between the shell and the core is expected to change.

To support the shell layer, the resource machine must have the functionality to produce, compose, and evaluate transaction candidates. Assuming the shell is otherwise trivial in the current version of the specification, the following description of the resource machine functionality describes the functionality of the resource machine core.

7.1. ARMs as intent machines. Together with $(CMtree, NFset)$, the Anoma Resource Machine forms an instantiation of the intent machine, where the state $S = (CMtree, NFset)$, a batch $B = Transaction$, and the transaction verification function of the resource machine corresponds to the state transition function of the intent machine as described in Hart and Reusche (2024). To formally satisfy the intent machine's signature, the resource machine's verify function may return the processed transaction along with the new state.

7.2. Create. Given a set of components required to produce a transaction, the create function produces a transaction data structure, which includes computing the nullifiers of the consumed resources, commitments of the created resources, transaction Δ and all of the required proofs.

We assume that the produced transaction induces a state change consuming resources $r_{i_1}, \cdots, r_{i_n}$ and creating resources $r_{o_1}, \cdots, r_{o_m}$.

Input:

- a set of $CMtree$ roots $\{rt_{i_k}, k \leq n\}$
- a set of resources $\{r_{i_1}, ..., r_{i_n}, r_{o_1}, ..., r_{o_m}\}$
- a set of nullifier secret keys $\{nk_{i_1}, ..., nk_{i_n}\}$
- extra data $extra$
- preference function Φ
- custom inputs required for resource logic proofs

Output: a transaction $tx = (rtscms, nfs, \Pi, \Delta_{tx}, extra, \Phi)$, where:

- $rts = \{rt_{i_1}, .., rt_{i_n}\}$
- $nfs = \{nf_{i_k} = h_{nf}(nk_{i_l}, r_{i_l}), k = 1..n\}$
- $cms = \{cm_{o_1} = h_{cm}(r_{o_l}), k = 1..m\}$
- $\Pi = \{\pi_{\Delta_{tx}}, \pi_{compl_1}, ..., \pi_{compl_c}, \pi_{i_1}, ..., \pi_{i_n}, \pi_{o_1}, ..., \pi_{o_m}\}$, where $1 \leq c \leq m + n$
- $\Delta_{tx} = \sum_k \Delta_{i_k} - \sum_l \Delta_{o_l}$
- $extra$
- Φ

7.3. Compose. Taking two transactions tx_1 and tx_2 as input produces a new transaction $tx = tx_1 \circ tx_2$ according to the transaction composition rules (6.3).

7.4. Verify. Taking a transaction as input verifies its validity according to the transaction validity rules (6.4). If the transaction is valid, the resource machine outputs a state update. Otherwise, the output is empty.

7.5. Stored data format. The ARM output state data that needs to be stored includes resource plaintexts, the commitment accumulator and the nullifier set. The table below defines the format for that data assumed by the ARM.

Name	Structure	Key Type	Value Type
Commitment accumulator (node)	Cryptographic accumulator	timestamp	$\mathbb{F}$
Commitment accumulator (leaf)	-	(timestamp, $\mathbb{F}$)	$\mathbb{F}$
Nullifier set	Set	$\mathbb{F}$	$\mathbb{F}$
Hierarchical index	Chained Hash sets	Tree path	$\mathbb{F}$
Data blob storage	Key-value store with deletion criteria	$\mathbb{F}$	(variable length byte array, deletion_criteria)

Table 2: Stored Data Format.

7.5.1. $CMtree$. Each commitment tree node has a timestamp associated with it, such that a lower depth tree node corresponds to a less specified timestamp: a parent node timestamp is a prefix of the child node timestamp, and only the leaves of the tree have fully specified timestamps (i.e. they are only prefixes of themselves). For a commitment tree of depth d, a timestamp for a commitment cm would look like $t_{cm} = t_1 : t_2 : .. : t_d$, with the parent node corresponding to it having a timestamp $t_1 : t_2 : .. : *$. The timestamps are used as keys for the key-value store. For the tree leaves, commitments are used along with the timestamps as keys. Merkle paths to resource commitments can be computed from the hierarchy of the timestamps.

7.5.2. $NFset$. Storing a nullifier set, nullifiers are used as keys in the key-value store. In future versions, a more complex structure that supports efficient non-membership proofs might be used for storing the nullifier set.

7.5.3. Hierarchical index. The hierarchical index is organised as a tree where the leaves refer to the resources, and the intermediate nodes refer to resource *subkinds* that form a hierarchy. The label of a resource r stored in the hierarchical index tree is interpreted as an array of *sublabels*: $r.label = [label_1, label_2, label_3, ...]$, and the i-th subkind is computed as $r.subkind_i = H_{kind}(r.l, r.label_i)$.

Remark 8. In the current version, only the subkinds derived from the same resource logic can be organized in the same hierarchical index path.

The interface of the tree enables efficient querying of all children of a specific path and verifying that the returned children are the requested nodes. Permissions to add data to the hierarchical index are enforced by the resource logics and do not require additional checks.

7.5.4. Data blob storage. Data blob storage stores data without preserving any structure. The data is represented as a variable length byte array and comes with a deletion criterion that determines for how long the data will be stored. The deletion criteria, in principle, is an arbitrary predicate, which in practice currently is assumed to be instantiated by one of the following options:

- delete after $block$
- delete after $timestamp$
- delete after sig over $data$
- delete after either predicate p_1 or p_2 is true; the predicates are instantiated by options from this list
- store forever

8. Program Formats

8.1. Transaction candidate. The system used to represent and interpret transaction candidates must have a deterministic computation model; each operation should have a fixed cost of space and time (for total cost computation). To support content addressing, it must have memory and support memory operations (read, write, allocate).

The system must support the following I/O operations:

- $READ\_STORAGE(address : \mathbb{F}_{cm})$: read the global content-addressed storage at the specified address and return the value stored at the address. If the value is not found, the operation should return an error. Storage not accessible to the machine accessing it will be treated as non-existent.
- $RESOURCES\_BY\_INDEX(index\_function)$: read resources in the history at execution time by the specified index function. If the index function output is invalid or uncomputable, or the resources cannot be located, the operation should return an error. Typically, the index functions allowed will be very restricted, e.g. an index function returning current unspent resources of a particular kind.

8.2. Gas model. To compute and bound the total cost of computation, the transaction candidate system must support a gas model. Each evaluation would have a gas limit g_{limit}, and the evaluation would start with $g_{count} = 0$. Evaluating an operation, the system would add the cost of the operation to the counter g_{count} and compare it to g_{limit}. When making recursive calls, g_{count} is incremented before the recursion occurs. If the value of g_{count} is greater than g_{limit}, the execution is terminated with an error message indicating that the gas limit has been surpassed.

8.3. Resource Logic. A resource logic is a predicate associated with a resource that checks that the input data satisfies a set of constraints. It does not require I/O communication and is represented by or can feasibly be turned into a zk-SNARK circuit.

Each resource logic has a set of public and private input values to support zk-SNARK representation. Resource logics are customizable on both implementation of the ARM (different instantiations might have different requirements for all resource logics compatible with this instantiation) and resource logic creation level (each instantiation supports arbitrary resource logics as long as they satisfy the requirements). A concrete implementation of the ARM can specify more mandatory inputs and checks (e.g., if the resources are distributed in-band, resource logics have to check that

DOI: 10.5281/zenodo.10498991

the distributed encrypted value indeed encrypts the resources created/consumed in the transaction), but the option of custom inputs and constraints must be supported to enable different resource logic instances existing on the application level.

The proving system used to interpret resource logics must satisfy the following properties:

- Verifiability. It must be possible to produce and verify a proof of type $PS.Proof$ that given a certain set of inputs, the resource logic output is true value.
- The system PS used to interpret resource logics must be zero-knowledge and function-privacy-friendly (support recursion, accumulation, or some other way to provide function privacy).

Resource logics take as input a subset of resources created and consumed in the transaction:

Resource Logic Public Inputs:

- $nfs \subset nfs_{tx}$
- $cms \subset cms_{tx}$
- custom inputs

Resource Logic Private Inputs:

- input resources corresponding to the elements of nfs
- output resource corresponding to the elements of cms
- $tag : \mathbb{F}_{tag}$ — identifies the resources being checked
- custom inputs

Resource Logic Constraints:

- for each output resource, check that the corresponding cm value is derived according to the rules specified by the resource machine instance
- for each input resource, check that the corresponding nf value is derived according to the rules specified by the resource machine instantiation
- custom checks

8.4. Preference Function. Preference functions do not require I/O communication or have any other special requirements. They are stateless. It may make sense to interpret them using the same system used for transaction candidates.

8.5. Nockma. Nockma (Nock-Anoma) is a modification of the Nock4K specification (Urbit) and a Nock standard library altered and extended for use with Anoma. Nockma is designed to support the transaction candidate interpreter requirements (8.1), namely, global storage read and deterministic bounded computation costs.

Nockma is parameterized over a specific finite field $\mathbb{F}_h$ and function h. The function h takes an arbitrary noun (a data unit in Nockma) as input and returns an element of $\mathbb{F}_h$. This function is used for verifying reads from content-addressed storage.

A *scry* (inspired by Urbit's concept of the same name) is a read-only request to Anoma's global content-addressed namespace or indexes computed over values stored in this namespace. Scrying is used to read data that would be inefficient to store in the noun, to read indexes whose value might only be known at execution time, or to read data that may not be accessible to the author of the noun.

Scrying comes in two types: “direct” or “index”. A direct lookup simply returns the value stored at the address (integrity can be checked using h), or an error if a value is not found. An index lookup uses the value stored at the address as an index function and returns the results of computing that index or an error if the index is not found, invalid, or uncomputable. The lookup type is the only parameter required apart from the content address (which must be an element of $\mathbb{F}_h$).

Typically, the index functions allowed will be very restricted, e.g. current unspent resources of a particular kind. Gas costs of scrying will depend on the index function and the size of the results returned.

Scrying may be used to avoid unnecessary, redundant transmission of common Nockma subexpressions, such as the standard library.

Nockma is a combinator interpreter defined as a set of reduction rules over nouns. A noun is an atom or a cell, where an atom is a natural number and a cell is an ordered pair of nouns.

Table 3 is an ordered list of reduction rules. The rules are applied from top to bottom, the first rule from the top matches. Variables match any noun. As in regular Nock4K, a formula that reduces to itself is an infinite loop, which we define as a crash (“bottom” in formal logic). A real interpreter can detect this crash and produce an out-of-band value instead.

The only difference between Nockma and Nock4K reduction rules is that instruction 12 is defined for scrying.

Pattern	Reduces to
$nock(a)$	$*a$
$[a\ b\ c]$	$[a\ [b\ c]]$
$?[a\ b]$	0
$?a$	1
$+[a\ b]$	$+[a\ b]$
$+a$	$1\ +\ a$
$=[a\ a]$	0
$=[a\ b]$	1
$/[1\ a]$	a
$/[2\ a\ b]$	a
$/[3\ a\ b]$	b
$/[(a\ +\ a)\ b]$	$/[2\ /[a\ b]]$
$/[(a\ +\ a\ +\ 1)\ b]$	$/[3\ /[a\ b]]$
$/a$	$/a$
$\#[1\ a\ b]$	a
$\#[(a\ +\ a)\ b\ c]$	$\#[a\ [b\ /[(a\ +\ a\ +\ 1)\ c]]\ c]$
$\#[(a\ +\ a\ +\ 1)\ b\ c]$	$\#[a\ [/[(a\ +\ a)\ c]\ b]\ c]$
$\#a$	$\#a$
$*[a\ [b\ c]\ d]$	$[*[a\ b\ c]\ *[a\ d]]$
$*[a\ 0\ b]$	$/[b\ a]$
$*[a\ 1\ b]$	b
$*[a\ 2\ b\ c]$	$*[*[a\ b]\ *[a\ c]]$
$*[a\ 3\ b]$	$?*[a\ b]$
$*[a\ 4\ b]$	$+*[a\ b]$
$*[a\ 5\ b\ c]$	$=[*[a\ b]\ *[a\ c]]$
$*[a\ 6\ b\ c\ d]$	$*[a\ *[[c\ d]\ 0\ *[[2\ 3]\ 0\ *[a\ 4\ 4\ b]]]]$
$*[a\ 7\ b\ c]$	$*[*[a\ b]\ c]$
$*[a\ 8\ b\ c]$	$*[[*[a\ b]\ a]\ c]$
$*[a\ 9\ b\ c]$	$*[*[a\ c]\ 2\ [0\ 1]\ 0\ b]$
$*[a\ 10\ [b\ c]\ d]$	$\#[b\ *[a\ c]\ *[a\ d]]$
$*[a\ 11\ [b\ c]\ d]$	$*[[*[a\ c]\ *[a\ d]]\ 0\ 3]$
$*[a\ 11\ b\ c]$	$*[a\ c]$
$*[a\ 12\ b\ c\ d]$	$result\ \leftarrow\ SCRY\ b\ c;\ *[a\ result\ d]$
$*a$	$*a$

Table 3: Nockma reduction rules.

Used with the resource machine, Nockma should return a set of modifications to the state transition expressed by the input transaction:

- a set of resources to additionally create (resource plaintexts)
- a set of resources to additionally consume (addresses)
- a set of storage writes (in the format specified in 7.5)

The Nockma standard library must include the following functions.
For a finite field $\mathbb{F}_n$ of order n, it should support:

- additive identity of type $\mathbb{F}_n$
- addition operation $\mathbb{F}_n \times \mathbb{F}_n \to \mathbb{F}_n$
- additive inversion $\mathbb{F}_n \to \mathbb{F}_n$
- multiplicative identity of type $\mathbb{F}_n$
- multiplication operation $\mathbb{F}_n \times \mathbb{F}_n \to \mathbb{F}_n$
- multiplicative inversion $\mathbb{F}_n \to \mathbb{F}_n$
- equality operation $\mathbb{F}_n \times \mathbb{F}_n \to \mathbb{F}_2$
- comparison operation based on canonical ordering $\mathbb{F}_n \times \mathbb{F}_n \to \mathbb{F}_2$

For a ring Z_n of unsigned integers $\bmod n$, it should support:

- additive identity of type Z_n
- addition operation $Z_n \times Z_n \to Z_n \times \mathbb{F}_2$ (with overflow indicator)
- subtraction operation $Z_n \times Z_n \to Z_n \times \mathbb{F}_2$ (with overflow indicator)
- multiplicative identity of type Z_n
- multiplication operation $Z_n \times Z_n \to Z_n \times \mathbb{F}_2$ (with overflow indicator)
- division operation (floor division) $Z_n \times Z_n \to Z_n$
- equality $Z_n \times Z_n \to \mathbb{F}_2$
- comparison $Z_n \times Z_n \to \mathbb{F}_2$

Additionally, it should provide a parametrized conversion function $conv_{i,j,k,l}$, where

- i is a flag that defines the input type: $i = 0$ corresponds to a finite field, $i = 1$ corresponds to a ring of unsigned integers
- j is the input structure order
- k is a flag that defines the output type: $k = 0$ corresponds to a finite field, $k = 1$ corresponds to a ring of unsigned integers
- l is the output structure order

If the order of the input structure is bigger than the order of the output structure ($j > l$), the conversion function would return a flag (of type $\mathbb{F}_2$) indicating if overflow happened in addition to the converted value.

The conversion function must use canonical ordering and respect the inversion laws.

9. Examples

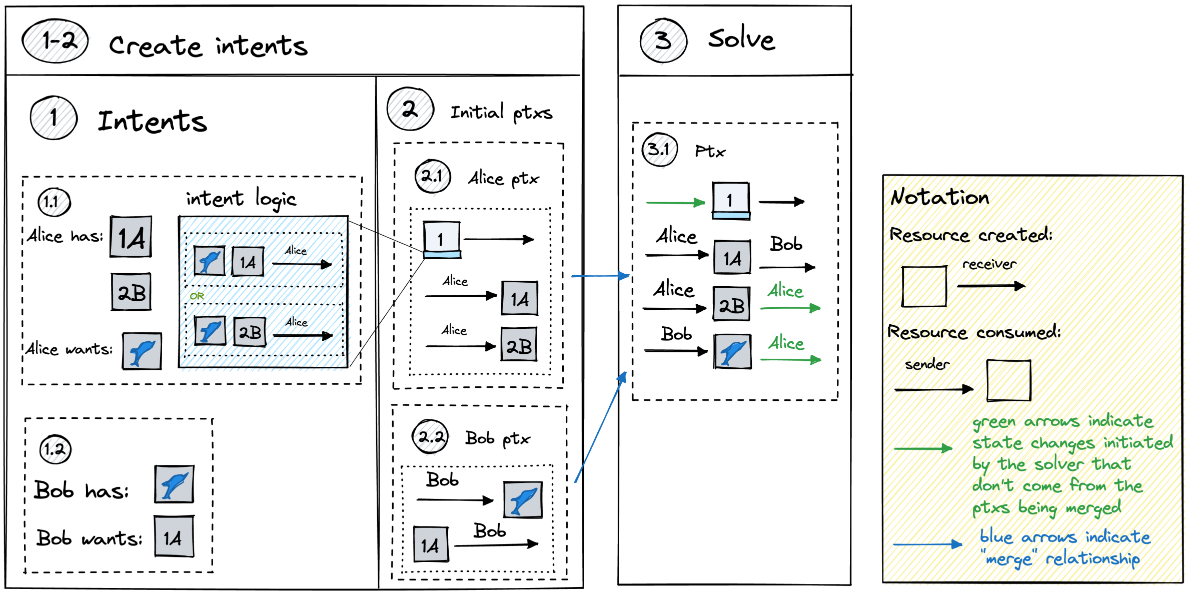
In the examples below, the superscript parameter indicates the party associated with the resource, e.g., a resource R^{Alice} is associated with Alice. In the case of proofs, it indicates the party created the proof.

9.1. Two Party Exchange.

Let us consider an example of a two-party exchange. One party's intent is precise, and the other party's intent implies options (the requested NFT's exact properties may also vary). We assume that the resources parties initially consume were already created at some point in the past.

Step 1: specify intents

- **Alice's intent:** exchange either a resource R_{1A} of kind A and quantity 1 or a resource R_{2B} of kind B and quantity 2 for a blue dolphin NFT resource R_{NFT}. The intent is contained in the resource logic of a resource R_I of kind I.
- **Bob's intent:** exchange a blue dolphin NFT resource for a resource of kind A and quantity 1. The intent is referred to in a transaction.



Remark 9. For simplicity, in the examples in this paper, the set of compliance proofs for initial transactions (the transactions that were not composed of other transactions) is assumed to contain all the necessary compliance proofs, but the proofs themselves are not specified. Additionally, the delta proof aggregation function can take an arbitrary number of arguments. $AGG(X, Y, Z)$ in practice would be implemented as $AGG(AGG(X, Y), Z)$, similarly defined for any number of proofs.

Step 2: create initial transactions
Alice creates a transaction TX^A creating R_I^A, and consuming R_{1A}^A and R_{2B}^A:

- $rts = \{rt_{R_{1A}^A}, rt_{R_{2B}^A}\}$
- $cms = \{cm_{R_I^A}\}$

- $nfs = \{nf_{R^A_{1A}}, nf_{R^A_{2B}}\}$
- Proofs:
  - Π^A_Δ
  - Π^A_{compl}
  - $\Pi^A_{rl} = \{\pi^A_A, \pi^A_B, \pi^A_I\}$
- $\Delta \mapsto \{I : 1, A : -1, B : -2\}$. For simplicity, represent Δ as a dictionary
- $extra = extra^A$
- $\Phi = \Phi^A$

Bob creates a transaction TX^B creating R^B_{1A} and consuming R^B_{NFT}:

- $rts = \{rt_{R^B_{NFT}}\}$
- $cms = \{cm_{R^B_{1A}}\}$
- $nfs = \{nf_{R^B_{NFT}}\}$
- Proofs:
  - Π^B_Δ
  - Π^B_{compl}
  - $\Pi^B_{rl} = \{\pi^B_A, \pi^B_{NFT}\}$
- $\Delta \mapsto \{NFT : -1, A : 1\}$
- $extra = extra^B$
- $\Phi = \Phi^B$

Step 3: solve
A solver S, having TX^A and TX^B, creates a transaction TX^S:

- $rts = \{rt_{R^A_I}\}$
- $cms = \{cm_{R^A_{2B}}, cm_{R^A_{NFT}}\}$
- $nfs = \{nf_{R^A_I}\}$
- Proofs:
  - Π^S_Δ
  - Π^S_{compl}
  - $\Pi^S_{rl} = \{\pi^S_B, \pi^S_{NFT}, \pi^S_I\}$
- $\Delta \mapsto \{NFT : 1, B : 2, I : -1\}$
- $extra = extra^S$

- $\Phi = \Phi^S$

and composes all three transactions together, producing a balanced transaction TX:

- $rts = \{rt_{R_I^A}, rt_{R_{2B}^A}, rt_{R_{1A}^A}, rt_{R_{NFT}^B}\}$
- $cms = cms^A \sqcup cms^B \sqcup cms^S = \{cm_{R_I^A}, cm_{R_{2B}^A}, cm_{R_{NFT}^A}, cm_{R_{1A}^B}\}$
- $nfs = nfs^A \sqcup nfs^B \sqcup nfs^S = \{nf_{R_I^A}, nf_{R_{2B}^A}, nf_{R_{1A}^A}, nf_{R_{NFT}^B}\}$
- Proofs:
  - $\Pi_\Delta = AGG(\Pi_\Delta^A, \Pi_\Delta^B, \Pi_\Delta^S)$
  - $\Pi_{compl} = \Pi_{compl}^A \sqcup \Pi_{compl}^B \sqcup \Pi_{compl}^S$
  - $\Pi_{rl} = \Pi_{rl}^A \sqcup \Pi_{rl}^B \sqcup \Pi_{rl}^S$
- $\Delta \mapsto \{A : 0, B : 0, I : 0, NFT : 0\}$
- $extra = extra^A \cup extra^B \cup extra^S$
- $\Phi = G(\Phi^A, \Phi^B, \Phi^S)$

In practice, the step of creation of the transaction TX_{S_1} can be merged with the composing step, but we separate the steps for clarity.

10. Future directions

This report contains the necessary information to build a resource machine that has the desired properties, but there are more properties we might want and more questions worth investigating. One such question would be whether resource logics should be able to see all resources in a transaction. This would allow us to perform “for all” checks — for example, a resource logic might want to enforce a non-inclusion of resources of a certain type in this transaction. However, enforcing such a feature is a non-trivial task, and it is not clear if it is as beneficial as it seems: for example, if there is a valid way to escape such checks (e.g., by wrapping a resource in another resource kind), it won’t be helpful to have a mechanism for checking.

References

Daira Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, 2023. URL https://zips.z.cash/protocol/protocol.pdf. (cit. on p. 3.)

Ilker Özçelik, Sai Medury, Justin T. Broaddus, and Anthony Skjellum. An overview of cryptographic accumulators. *CoRR*, abs/2103.04330, 2021. URL https://arxiv.org/abs/2103.04330. (cit. on p. 4.)

Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. 2023. URL https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf. (cit. on p. 6.)

Anthony Hart and D Reusche. Abstract Intent Machines. *Anoma Research Topics*, February 2024. doi:10.5281/zenodo.10498993. URL https://doi.org/10.5281/zenodo.10118865. (cit. on p. 9.)

Urbit. Nock definition. URL https://docs.urbit.org/language/nock/reference/definition. (cit. on p. 12.)

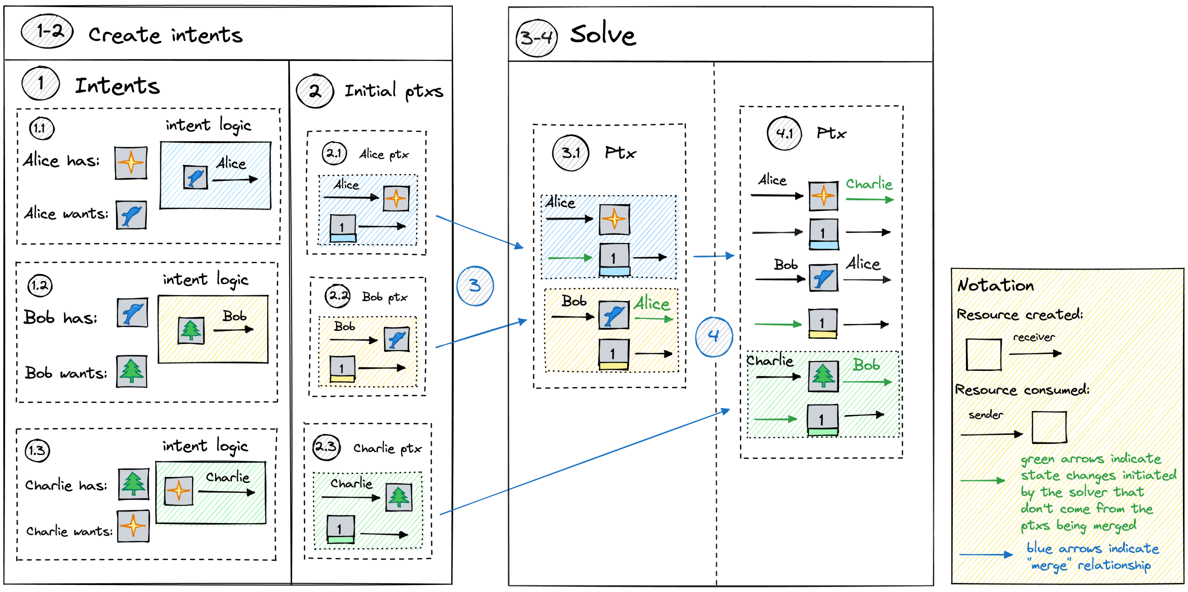
A. Three-party NFT exchange cycle

Another example is a three-party exchange cycle. Each party uses ephemeral resource logics to express their intents.

Step 1: specify intents

- **Alice’s intent:** Alice wants to exchange her star NFT resource R_{star}^A for a blue dolphin NFT resource $R_{dolphin}$

- **Bob's intent:** Bob wants to exchange his blue dolphin NFT $R^B_{dolphin}$ for a tree NFT resource R_{tree}
- **Charlie's intent:** Charlie wants to exchange his tree NFT R^C_{tree} for a star NFT resource R_{star}



Step 2: create initial transactions
Alice's initial transaction:

- $rts = \{rt_{R^A_{star}}\}$
- $cms = \{cm_{R^A_{I^A}}\}$
- $nfs = \{nf_{R^A_{star}}\}$
- Proofs:
  - Π^A_Δ
  - Π^A_{compl}
  - $\Pi^A_{rl} = \{\pi^A_{star}, \pi^A_I\}$
- $\Delta \mapsto \{I^A : 1, star : -1, \}$ – for simplicity, represent Δ as a dictionary
- $extra = extra^A$
- $\Phi = \Phi^A$

Bob's initial transaction:

- $rts = \{rt_{R^B_{dolphin}}\}$
- $cms = \{cm_{R^B_{I^B}}\}$
- $nfs = \{nf_{R^B_{doplhin}}\}$
- Proofs:

DOI: 10.5281/zenodo.10498991

  - Π^B_Δ
  - Π^B_{compl}
  - $\Pi^B_{rl} = \{\pi^B_{dolphin}, \pi^B_I\}$

- $\Delta \mapsto \{I^B : 1, dolphin : -1\}$
- $extra = extra^B$
- $\Phi = \Phi^B$

Charlie's initial transaction:

- $rts = \{rt_{R^C_{tree}}\}$
- $cms = \{cm_{R^C_{I^C}}\}$
- $nfs = \{nf_{R^C_{tree}}\}$
- Proofs:
  - Π^C_Δ
  - Π^C_{compl}
  - $\Pi^C_{rl} = \{\pi^C_{tree}, \pi^C_I\}$
- $\Delta \mapsto \{I^C : 1, tree : -1, \}$
- $extra = extra^C$
- $\Phi = \Phi^C$

Step 3: solve
A solver S_1, seeing TX^A and TX^B, creates a transaction TX^{S_1} (on the diagram: $TX_{3.1}$, green arrows):

- $rts = \{rt_{R^A_{I^A}}\}$
- $cms = \{cm_{R^A_{dolphin}}\}$
- $nfs = \{nf_{R^A_{I^A}}\}$
- Proofs:
  - $\Pi^{S_1}_\Delta$
  - $\Pi^{S_1}_{compl}$
  - $\Pi^{S_1}_{rl} = \{\pi^{S_1}_{dolphin}, \pi^{S_1}_{I_A}\}$
- $\Delta \mapsto \{dolphin : 1, I^A : -1\}$
- $extra = extra^{S_1}$
- $\Phi = \Phi^{S_1}$

and composes all three transactions together, producing a transaction $TX_{3.1}$:

- $rts = \{rt_{R^A_{star}}, rt_{R^B_{dolphin}}, rt_{R^A_{IA}}\}$
- $cms = cms^A \sqcup cms^B \sqcup cms^{S_1} = \{cm_{R^A_{IA}}, cm_{R^B_{IB}}, cm_{R^A_{dolphin}}\}$
- $nfs = nfs^A \sqcup nfs^B \sqcup nfs^{S_1} = \{nf_{R^A_{star}}, nf_{R^B_{dolphin}}, nf_{R^A_{IA}}\}$
- Proofs:
  - $\Pi^{3.1}_{\Delta} = AGG(\Pi^A_{\Delta}, \Pi^B_{\Delta}, \Pi^{S_1}_{\Delta})$
  - $\Pi^{3.1}_{compl} = \Pi^A_{compl} \sqcup \Pi^B_{compl} \sqcup \Pi^{S_1}_{compl}$
  - $\Pi^{3.1}_{rl} = \Pi^A_{rl} \sqcup \Pi^B_{rl} \sqcup \Pi^{S_1}_{rl}$
- $\Delta \mapsto \{I^A : 0, I^B : 1, star : -1, dolphin : 0\}$
- $extra = extra^A \cup extra^B \cup extra^{S_1}$
- $\Phi = G(\Phi^A, \Phi^B, \Phi^{S_1})$

Step 4: continue solving

Seeing TX^C and $TX_{3.1}$, a solver S_2 creates a transaction TX_{S_2} (on the diagram: $TX_{4.1}$, green arrows):

- $rts = \{rt_{R^C_{IC}}, rt_{R^B_{IB}}\}$
- $cms = \{cm_{R^C_{star}}, cm_{R^B_{tree}}\}$
- $nfs = \{nf_{R^C_{IC}}, nf_{R^B_{IB}}\}$
- Proofs:
  - $\Pi^{S_2}_{\Delta}$
  - $\Pi^{S_2}_{compl}$
  - $\Pi^{S_2}_{rl} = \{\pi^{S_2}_{star}, \pi^{S_2}_{I_C} \pi^{S_2}_{tree}, \pi^{S_2}_{I_B}\}$
- $\Delta \mapsto \{I^C : -1, I^B : -1, star : 1, tree : 1\}$
- $extra = extra^{S_2}$
- $\Phi = \Phi^{S_2}$

and composes all three into a balanced transaction $TX_{4.1}$:

- $rts = \{rt_{R^A_{star}}, rt_{R^B_{dolphin}}, rt_{R^C_{tree}}, rt_{R^A_{IA}}, rt_{R^B_{IB}}, rt_{R^C_{IC}}\}$
- $cms = cms^{TX^{3.1}} \sqcup cms^{S_2} = \{cm_{R^A_{dolphin}}, cm_{R^B_{tree}}, cm_{R^C_{star}}, cm_{R^A_{IA}}, cm_{R^B_{IB}}, cm_{R^C_{IC}}\}$
- $nfs = nfs^{TX^{3.1}} \sqcup nfs^{S_2} = \{nf_{R^A_{star}}, nf_{R^B_{dolphin}}, nf_{R^C_{tree}}, nf_{R^A_{IA}}, nf_{R^B_{IB}}, nf_{R^C_{IC}}\}$
- Proofs:

  - $\Pi_{\Delta}^{4.1} = AGG(\Pi_{\Delta}^{3.1}, \Pi_{\Delta}^{C}, \Pi_{\Delta}^{S_2})$
  - $\Pi_{compl}^{4.1} = \Pi_{compl}^{C} \sqcup \Pi_{compl}^{3.1} \sqcup \Pi_{compl}^{S_2}$
  - $\Pi_{rl}^{4.1} = \Pi_{rl}^{C} \sqcup \Pi_{rl}^{3.1} \sqcup \Pi_{rl}^{S_2}$
- $\Delta \mapsto \{I^A : 0, I^B : 0, I^C : 0, star : 0, dolphin : 0, tree : 0\}$
- $extra = extra^A \cup extra^B \cup extra^C \cup extra^{S_1} \cup extra^{S_2}$
- $\Phi = G(\Phi^A, \Phi^B, \Phi^{S_1}, \Phi^C, \Phi^{S_2})$

In practice, the step of creation of the transactions TX_{S_1} and TX_{S_2} can be merged with the composing step, but we separate the steps for clarity.

DOI: 10.5281/zenodo.10498991