Agora DRep Effect Specification

Tomasz Maciosowski
tomasz@mlabs.city

Gergely Szabó
gergely@mlabs.city

Abstract

DAOs that have large treasuries of ADA tokens under their control cannot use that to directly participate in Cardano-level protocol governance introduced in the Chang hard fork. They would need to delegate their stake to a dRep and rely on them as a centralized party that should reflect the values of the whole organization. Instead, we propose to create a dRep onchain validator. DAOs should be able to delegate their treasuries to dRep validators that they directly control. Using Agora proposals, organization members will be able to decide how their DAO will vote instead.

1 Overview

This document provides a specification of a set of Cardano validators that together provide Agora effect that allows DAO members to use Agora voting system to decide how they want to act on Cardano-level governance proposal together as an organization. Plutus V3 script can be registered as a Cardano DRep and we will use that to create a Plutus DRep that will allow voting on Cardano proposals only after the organization has passed a successful "internal" vote on that decision using their Agora instance. First iteration of the DRep Effect will be limited in governance actions it is allowed to do only to voting, i.e. it will not be able to create Cardano governance proposals. The implementation of this specification will be done in Plutarch and available on GitHub.

1.1 V2 / V3 Interactions Limitations

The Cardano ledger imposes limitations on what transaction can do depending on which versions of Plutus scripts are being executed in the same transaction. The limitation that is relevant for us is that transactions that execute V2 Plutus scripts cannot also contain Cardano governance voting actions (see Table 1).

Feature	Plutus V2	Plutus V3	Plutus V2 + V3
Plutus V1 Primitives	Yes	Yes	Yes
Governance Voting	No	Yes	No

Table 1: Part of Cardano Script Compatibility Chart

The Plutus V2 script in question is the minting policy of the Governance Authority Tokens (GAT) of already deployed Agora instances. It is important for organizations to continue using

their current Agora instances so rather then requiring them to re-deploy Agora using Plutus V3 we will provide a GAT migration mechanism - a proxy validator that will convert GATs from V2 to V3 on demand. More over the migration mechanism is separate from the voting effect thus may be used for other effects that need access to transaction features that were introduced after Plutus V2.

2 Validators

2.1 Proxy Spending Validator

Plutus V2 Agora instances can use this validator to use effects that are implemented in newer Plutus versions. To do so instead of specifying actual effect in the proposal datum the user can specify this validator instead and specify the final V3 GAT destination using datum of the Proxy Spending Validator. As a security measure the datum allows only for script destinations to make sure that GAT never reaches a public key address as it may compromise the whole Agora system. Proxy Spending Validator is parametrized by the real Agora Governance Authority Token symbol and allows for spending only if it's UTxO contains that GAT and the transaction burns it.

Note that the actual minting will be done by Proxy Minting Validator. While they are described as two separate validators actually they will be the same onchain script that acts differently depending on the purpose it is spent with. That approach avoids circular script parametrization as minting part needs to know the address of the spending part and spening part needs to know the currency symbol of the minting part, if they are implemented in the same script that then the script hash is the same and it is passed as a runtime parameter from Cardano node.

For the sake of simplicity and correctness the implementation is limited to migrating only one GAT per transaction. In practice such limitation is irrelevant as most Agora proposals have only one effect in the outcome map and proposals are rare enough that the cost of submitting multiple transactions is irrelevant.

Note that this validator is in fact an Agora effect thus the spending conditions must follow the restrictions for Agora effects such as not doing anything that is irrelevant to the GAT migration process as that may accidently allow to trigger other action that is gated behind burning V2 GAT.

Data Types

```
data Datum = Datum
  { receiverScript :: ScriptHash
  , datumHash :: DatumHash
  }
```

Spending Conditions

- Transaction burns one GAT (symbol is known from script parameter)
- Spent UTxO contains GAT
- Transaction creates a UTxO at address of `receiverScript` with empty staking part and reference datum with hash equal to `datumHash`.
- Transaction does not mint or burn any tokens other than V2 and V3 GATs.

- Transaction does not include any certificates.
- Transaction does not include script inputs other than own input.

2.2 Proxy Minting Validator

Minting validator is tightly coupled to the spending validator and some checks are offloaded there. This validator succeeds when the transaction spends from the spending validator address (that already ensures proper handling of V2 GAT) and that it mints exactly one V3 GAT with empty token name.

Spending Conditions

- Transaction contains an input from Proxy Spending Validator. The address is known as Minting and Spending validators share the same hash.

2.3 Spending Effect Validator

Because onchain voting action is gated behind burning pGAT the effect implementation has to be split into spending and voting validators. Spending validator simply allows only for burning the pGAT as long as the transaction.

Spending Conditions

- Transaction contains a vote with own drep hash

Rest of the checks if performed by the Voting Effect validator.

Interaction diagrams

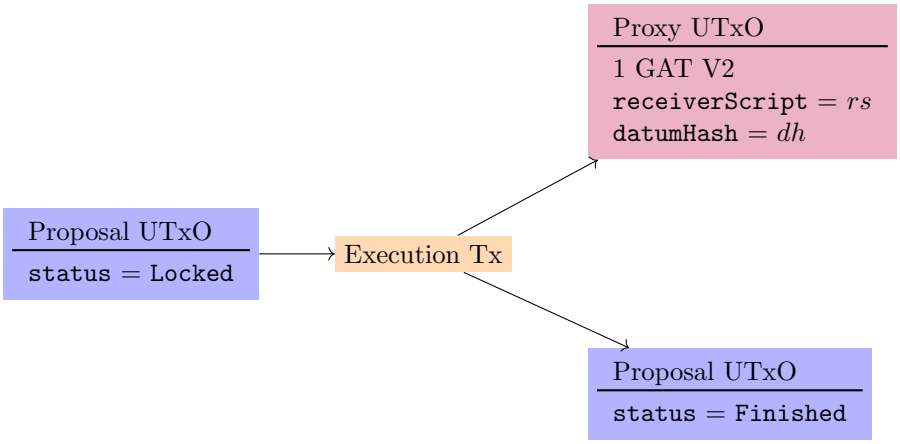

Figure 1: Minting V2 GAT

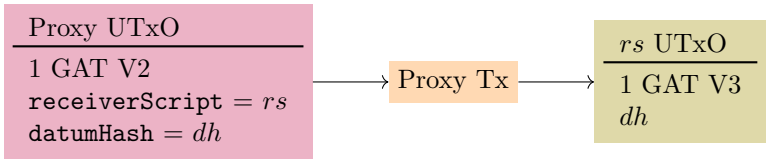

Figure 2: GAT Upgrade to V3

2.4 Voting Effect Validator

The Voting Effect validator will utilise the above established Proxy Minting Validator, minting a Proxy Governance Authority Token (pGAT). By spending the pGAT, we essentially get the same guarantees as with GAT, but we can use Plutus V3, which introduces the functionalities to work with native Cardano governance. Similarly to the Proxy Validator, the script serves multiple purposes, to avoid the circular dependency problem (see explanation in 2.1 Proxy Spending Validator).

In order to vote on a Cardano Governance Action using this effect the following preconditions must be met:

- The Voting Effect validator must be registered as a DRep
- staking verification key of the DAO should be delegated to the above DRep
- Governance action with well known id must be already submitted on-chain

The effect transaction will burn exactly one pGAT, and at the same time place a vote to the governance action defined in the datum.

DRep registration

Registering the Voting Effect validator is a one time action, that has to be done after deployment of the protocol. The validator script will allow DRep registration unconditionally, but DRep update will be prohibited.

Data Types

```
data Datum = Datum
  { governanceActionId :: GovernanceActionId
  , vote :: Vote
  }
```

Spending Conditions

- `ScriptPurpose` is either `Certifying`, `Voting`, or `Spending`
  - if `Spending`
    - `ScriptContext` contains exactly 1 script input
    - `ScriptContext` contains exactly 1 vote where the `Voter` is the DRep credential of the Voting Script (script hash is common with the spending script)
  - if `Certifying`

        * `ScriptContext` contains exactly 1 transaction DRep registration certificate
        * `ScriptContext` does not contain a vote
        * `ScriptContext` does not contain proposal procedures
        * `ScriptContext` does not contain treasury donations
    - if `Voting`
        * above input has `Voting Effect datum` as a hashed datum
        * `ScriptContext` contains exactly 1 vote, where the `GovernanceId` and `Vote` match the ones in the `Voting Effect's datum`
        * `ScriptContext` does not contain a certificate
        * `ScriptContext` does not contain proposal procedures
        * `ScriptContext` does not contain treasury donations

Interaction Diagrams

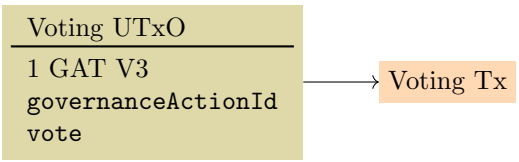

Figure 3: Voting

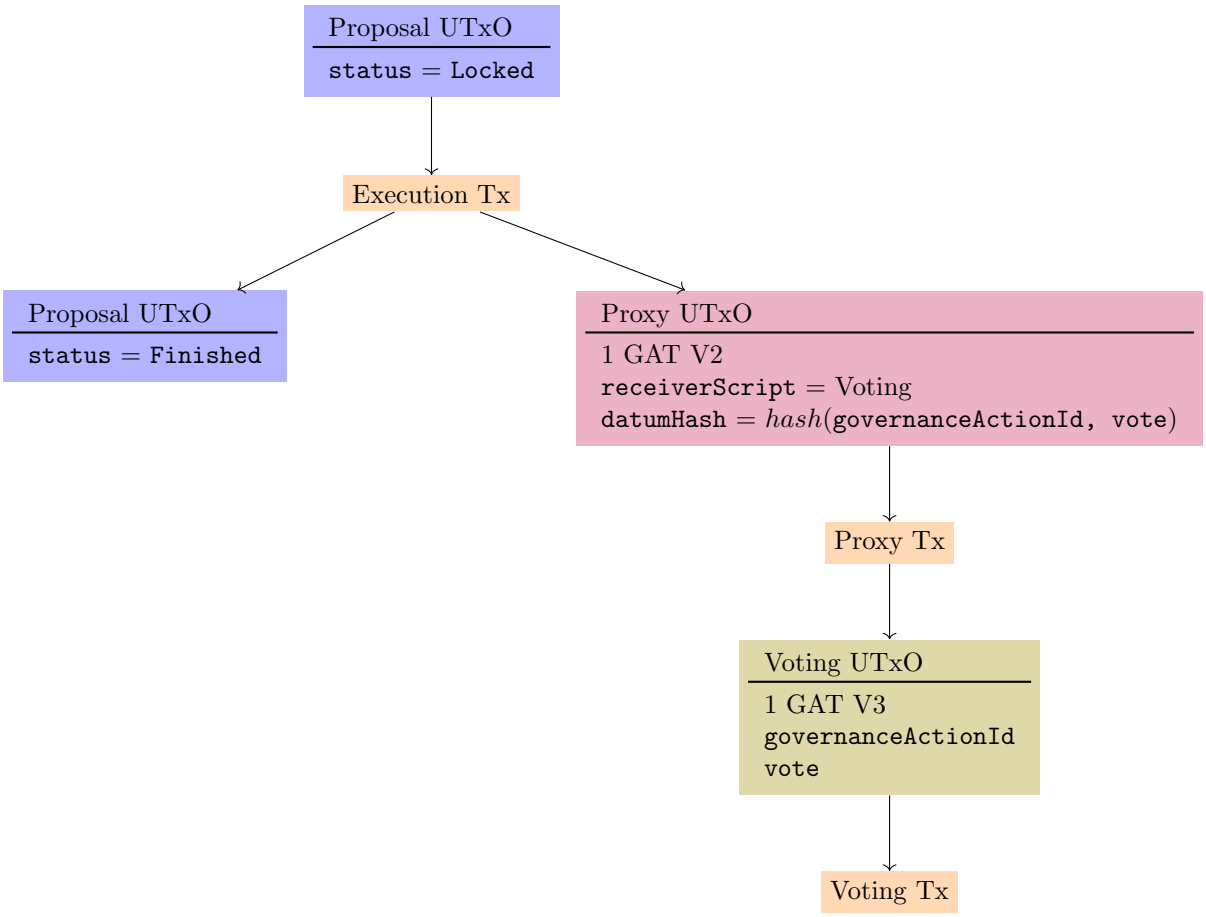

Figure 4: Full Workflow

3 Data Encoding

Datums and redeemers defined in this specification have to be data-encoded using the following rules in order to achieve the optimal encoding:

- Enums (SoPs where all constructors have no fields) have to be encoded as an integer starting from 0 following the constructor order from the specification.
- Records (SoPs with only one constructor with at least one field) have to be encoded as a list of data encoded fields rather than wrapped with `Constr 0`.
- Everything else has to be encoded using `Constr` with index starting from 0 following the constructor from the specification and field list in order of the declaration.

Note that we do not "mix" the encodings i.e. even if one of the constructors of the generic SoP has no fields it still has to be encoded using `Constr` rather than as an integer. Types that are not defined in this document are standard Plutus types with encoding already specified and the encoding rules do not apply to them.

When the specification does not describe datum or redeemer it is assumed to be never accessed thus plutus data passed to the script is irrelevant and the offchain implementation is free to pass anything and the validator has to succeed anyway. Usually the offchain transaction building code will pass unit type encoded as `Constr 0 []` or `0` however it is not strictly required to do so.

3.1 Plutarch Details

Note that the following encodings can be achieved automatically using Plutarch's derivation strategies, namely:

- Enum encoding can be derived with `DeriveAsTag` for both `PlutusType` and `PLiftable`
- Record with `DeriveAsDataRec` for `PlutusType` and `DeriveDataPLiftable` for `PLiftable`
- Generic SoP encoding can be derived with `DeriveAsDataStruct` for `PlutusType` and `DeriveDataPLiftable` for `PLiftable`

4 Estimate

- Milestone 1 (Spec): 199 hours
  - 75 h - Research spending, minting, and voting conditions
  - 62 h - Draft specifications for each validator
  - 62 h - Create a detailed LaTeX document for the conditions
- Milestone 2 (Proxy Implementation): 182 hours
  - 62 h - Develop the proxy spending validator using Plutarch
  - 62 h - Implement the V3 minting policy
  - 58 h - Set up a GitHub repository for the project
- Milestone 3 (Spending/Voting Implementation): 182 hours
  - 68 h - Create a comprehensive test suite for the proxy validator
  - 54 h - Conduct code reviews
  - 46 h - Validate the implementation using test cases
  - 14 h - Document results in a test report
- Milestone 4 (Off-chain & Frontend Integration): 234 hours
  - 89 h - Develop the spending effect validators in Plutarch
  - 74 h - Write test cases for the spending validators
  - 57 h - Review and validate the code with tests
  - 14 h - Upload the code to GitHub
- Milestone 5 (Documentation): 234 hours
  - 90 h - Develop the voting effect validators in Plutarch

  - 58 h - Write and execute test cases for voting validators
  - 58 h - Conduct final code reviews and validation
  - 28 h - Ensure documentation and upload the code to GitHub
- Milestone Final (End-to-End Testing): 300 hours
  - 101 h - Develop off-chain components for clarity.vote integration
  - 71 h - Implement frontend interface for proposals and voting
  - 71 h - Create API documentation and tutorial guides
  - 57 h - Test on Testnet and prepare a closeout report with a demo video

Total: 1331 h