SMV contract verification report (Stage1)
May 31, 2021
Pruvendo team

Executive Summary

This document describes the high-level approach, methodology, and the business-level scenarios for the verification of the central SMV smart contracts. The purpose of the document is as follows:

- Provide a formal description of the central SMV system's smart contract behavior in different cases
- Identify possible malfunctions and attacks
- Formally describe how to formally verify (prove) that the behavior of the smart contracts is precisely the same as one described in the document, and that expected attacks are not possible

Reference implementation

The reference implementation used for the present specification is:

- Location - https://github.com/RSquad/BFTG
- Hash - b054d45d04ec9f04769b8e1e2e692b022234eb79
- Branch - master

A brief description of the SMV contracts

The SMV contracts implement the soft majority voting algorithm envisioned in the contest proposal. The idea is that the voting doesn't necessitate the participation of the actual majority of the Free TON community - rather, the intention is that there is a sufficient number of votes envisioned at the inception of a contest, and the voting can be ended if a sufficient number of votes have been cast, thus streamlining the voting and approval process.

The smart contracts investigated in the course of the current contest are as follows:

- Demiurge: this is the central contract which deploys Padawans (smart contracts that accumulate votes from users and instantiate the casting of the votes)
- Padawan: these are smart contracts deployed by Demiurge that accumulate funds and cast votes for Proposals
- Proposal: these are smart contracts deployed by Demiurge that accumulate Padawan votes and, depending on the options set out at a contest's inception, decide on the voting's ending and send the result of the voting back to the Demiurge contract.

It worth mentioning that the reference implementation of SMV is a part of larger BFTG contract and for the purposes of the present specification the code and API not related to SMV is ignored and is not taken into consideration. Also no debots are considered limiting the number of contracts being specified to three: *Demiurge*, *Padawan* and *Proposal*.

Foundations

The Curry-Howard Isomorphism

The foundation of our approach to the formal verification of smart contracts is the **Curry-Howard isomorphism[1].** The isomorphism maps programs to (constructive) proofs in logic (and vice versa). This, in turn, means that we can *prove* programs, i. a. using proof assistants, to investigate and verify them.

Imperative vs Functional, eDSLs, and Syntactic Equivalence

The program / smart contract that we investigate must satisfy certain conditions / be written in a language of a specific kind so that we can make use of the Curry-Howard isomorphism.

Smart contracts are usually written in *imperative* languages, such as C++ or Solidity, while proof assistants use *functional* languages to specify and prove programs.

Therefore, before we proceed with the formal verification, the smart contract must be first translated from the original language into a purpose-built *embedded domain-specific language (eDSL)*. Using a state-monadic model, an eDSL allows us to implement imperative statements in a functional environment so that the proof assistant can handle them.

Using an eDSL allows us to maintain *syntactic equivalence* with the original code in an imperative language[2].

The original program and the product of the translation (the eDSL program) are syntactically equivalent if:

1. One program can be translated into another without losing any data (with the exception of comments, spaces, tabulation symbols etc.)
2. The translation in at least one direction can be accomplished using “simple” tools such as regular expressions, while the generic syntactic structure remains unchanged

---

[1] Howard, William A. (September 1980), "The formulae-as-types notion of construction", in Seldin, Jonathan P.; Hindley, J. Roger (eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, pp. 479–490, ISBN 978-0-12-349050-6.)

[2] Please note that complete syntactic equivalence is sometimes hard to achieve due to the syntax restrictions of the (functional) general-purpose language into which the DSL is embedded

3. From a human's point of view, both programs are structured and can be read in a rather similar way.

The Coq approach to formal verification

4.1. Introducing Coq

Our formal verification solution employs *Coq Proof Assistant* (also referred to as simply Coq). First introduced in 1989, Coq has been actively developed and used for many theoretical and practical applications ever since.

Coq is a formal proof management system that is based on a pure mathematical paradigm called *Calculus of constructions*[3] (and its extension called *Calculus of Inductive Constructions*[4]) that allows to use mathematical induction[5] in addition to pure formal logic[6].

The Coq distribution includes an OCaml-like formal specification language called *Gallina* for mathematical definitions, executable algorithms, and theorems together with an environment for semi-automatic development of machine-checked proofs.

As a proof development system, Coq provides interactive proof methods, decision and semi-decision algorithms. Also, Coq comes with a variety of predefined *tactics* (proof methods, corresponding to steps in the conventional proving process) and a dedicated language, called L_{tac}, that can be used to define custom tactics.

Finally, Coq packs a graphical tool, CoqIDE, that allows a user to navigate the Coq file forward and backward, executing and/or undoing specific commands, and an API that allows to integrate Coq with various development environments (e.g. Microsoft VSCode).

(We can state that) Coq is a powerful and reliable proof assistant. That is, if Coq concludes that a statement is proved, we can safely assume that it is indeed proven.

4.2. Smart contracts and the Coq toolbox

As we have mentioned in the *Foundations* section, a smart contract is usually written in an imperative language, whereas Gallina, the Coq specification language, is a functional one.

We use the following tools to convert the original imperative code to Gallina:

---

[3] https://hal.inria.fr/file/index/docid/76024/filename/RR-0530.pdf
[4] https://coq.inria.fr/distrib/current/refman/language/cic.html
[5] https://encyclopediaofmath.org/index.php?title=Mathematical_induction
[6] http://www.collegepublications.co.uk/logic/mlf/?00029

- Dedicated Gallina-based **embedded domain-specific languages** (eDSLs) that allow to describe the statements originally written in their respective imperative language (C++ or Solidity) in terms of Gallina
- A **translator** that automatically converts the code written in a specific imperative language into the respective eDSL

The both eDSLs and the translator are proprietary products of the Pruvendo team.

After these tools are applied, we can explicitly specify and verify the original smart contract code with Coq alone **and/or** using *FinProof Base*, a set of purpose-built proprietary libraries.

Note: There are certain limitations pertaining to the original imperative code as far as automatic code conversion is concerned. The fundamental limitation, based on the *Turing halting problem*[7] is that all cycles must have strong evidence of terminating at some point. To honor this limitation, developers are highly encouraged to use weak normalization (*or roughly, with decreasing explicit iterator variables*). If this limitation is not accounted for, the code conversion ceases to be automatic, requiring significant, non-trivial manual interventions so that the resulting eDSL code is accepted by *Coq*.

Coq and the TON ecosystem

When we begin the formal verification of a smart contract with Coq in the above-described way, we must assume that the TON blockchain and the the TON Virtual Machine (TVM) work in accordance with their respective specifications.

Caveat: As far as one of the TON Compilers is concerned, we may still assume it is not sufficiently reliable, if this must be accounted for.

The formal verification of the TON Compiler has been out of scope for the time being. If necessary, the TON Compiler can be formally verified using our purpose-built Coq-Based *TON Reference Virtual Machine (CTRVM). CTRVM* was developed strictly according to the TVM specification, and it has been formally verified for the most part (certain parts thereof, such as hashmaps, are still in the works, but this shouldn't preclude one from using *CTRVM*).

A semi-automated runner can run the contract being investigated and ensure that the TVM code, compiled by the TON Compiler and executed by *CTRVM*, works exactly in the same manner as the original imperative code of the smart contract.

4.4. Further assumptions: a checklist

In addition to the above-mentioned assumptions regarding the TON ecosystem, we also make the following assumptions before we proceed with the formal verification of a smart contract:

[7] Alan Turing, On computable numbers, with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Series 2, Volume 42 (1937), pp 230–265, doi:10.1112/plms/s2-42.1.230

- When a certain statement is not explicitly specified, we can make the necessary assumptions on a common-sense basis
- Coq Proof Assistant or any other tool we use works correctly
- Infinite sequences of similar elements (arrays) are allowed
- Direct or indirect recursions (function-based or message-based) are not allowed
- Each cycle must exit at some predefined point (no infinite loops allowed)

Smart Contract Specification: General Considerations

The finite state machine approach

After a set of experiments, we have chosen the finite state machine as a basic tool for scenario building. However, the full set of states is too big to be handled. To overcome this, we use the *projections* of states to a rather small set of hypersurfaces to produce the scenarios.

It is important to ensure that the selected set of hypersurfaces is *comprehensive*, i.e. it covers *all* the dimensions of the original state *of a given smart contract*. This allows us to restrict the overall number of scenarios to a *sum* of scenarios for each hypersurface rather than the *product* thereof, as this would be for the full set of states.

When describing a hypersurface, we use the conventional finite state machine graphic notation:

- rectangles depict distinct states
- arrows depict the transitions between the states
- arrow titles depict transition conditions (note that the conditions are heterogeneous as they usually comprise both an external event and a logical condition)

We also use state machine projections when we develop business-level / user scenarios.

The formalized natural language for smart contract specification

To provide a specification of a smart contract, we use a *formalized natural language*. This means that each expression can be literally (presumably, manually) translated into a formal computer language. Currently the formalized expressions are rather flexible but in future it's planned to strictly specify such a formalized language and implement an automatic translator into *Coq*.

The following meanings of English words (as capitalized) are used (as well as others):

- NO - means that the whole statement is always *True*
- POSITIVE - an integer that is greater than 0 and less than maximum value of the specified type such as 2^{64} or 2^{128}

- MUST - a mandatory requirement, can be translated as an ∀ quantifier for the left side to the right
- EMPTY - a list with no elements
- EQUAL(S) - the left part of the expression is fully equivalent to the right one (has the same type and all the values are structurally EQUAL). As an example 1 is not EQUAL to 1.0 .
- UNCHANGED - the projection of the state to the hypersurface described by the element after the completion of the function (or a sequence of functions) being described EQUALS to the corresponding projection before the call
- NO CHANGES OCCUR - the state is UNCHANGED (note: this doesn't cover gas, balance or any other non-contract specific states if they are considered to be a part of the state)
- ARE NOT MET - implication from the negation of the left part to the right part
- ZERO - integer zero
- UNIQUE IN … IN TERMS OF - No element of mapping of the middle statement by the right statement EQUALS to the left statement
- MUST EXIST IN … IN TERMS OF - Some element of mapping of the middle statement by the right statement EQUALS to the left statement
- EXISTING IN … IN TERMS OF - The first element of mapping of the middle statement by the right statement EQUALS to the left statement
- LESS OR EQUAL - corresponds to <=
- MINUS - corresponds to the arithmetic subtraction
- BEING ADDED WITH - the right statement is added to the list as the first element, other elements are UNCHANGED
- AND - conjunction of two statements
- OR - disjunction of two statements
- NON NEGATIVE - ZERO OR POSITIVE
- IN CASE OF SUCCESS … OTHERWISE - if no exceptions occur in the whole tree of messages as well as no messages are bounced the middle expression MUST be true, otherwise the right expression MUST be true
- INCREASED BY - After the execution of function or set of functions the left expression is larger that its value before the execution by the right expression
- OF - the value of the left expression of the structure represented by the right expression
- IF … THEN … OTHERWISE - self-explained construction

Note: In addition to the above-listed generic expressions, purpose-built expressions that refer to the properties of a given smart contract may be introduced. These shall be included in the specification of that respective contract after the generic ones.

In case the formalized description becomes too bulky the new definitions may be added (the sets of such definitions are called "special stories").

As I result usage of such an approach allows to accomplish two goals that are often considered as controversial:

- Specification must be readable and understandable by the regular IT-professionals and product owners
- Specification must be easily converted into a formal language

Specifications of the SMV smart contracts under consideration

Types and interfaces

For the convenience of further reading, the following types and interfaces are introduced as some of them are used by the cross-contract API (aka public methods/functions/messages) and the others are applied for the representation of internal states of the certain contracts. Also an external contract, *DemiurgeStore,* must be taken into account as the eligible provider of the key state information for the Demiurge contract.

Constants

The following constants are defined throughout the project to simplify the understanding of further text.

Exceptions

Name	Value	Description
ERROR_NOT_AUTHORIZED_WALLET	300	Raised when a wallet other than owner calls a *Padawan*
ERROR_PADAWAN_ALREADY_DEPLOYED	301	Raised when the *Padawan* for the specified wallet already exists
ERROR_PROPOSAL_ALREADY_DEPLOYED	302	Raised when the *Proposal* for the specified wallet already exists
ERROR_NOT_ALL_CHECKS_PASSED	303	Raised when some data for the *Demiurge* is not provided yet
ERROR_INIT_ALREADY_COMPLETED	304	Raised when somebody tries to reinitialize *Demiurge*
ERROR_UNAUTHORIZED_CALLER	110	Raised when somebody but parent *Demiurge* tries to initialize *Padawan*
ERROR_NOT_ENOUGH_VOTES	111	Raised when attempt to use more votes than available

		occurs
ERROR_NOT_A_USER_WALLET	113	Raised when somebody but user wallet tries to invoke specific *Padawan* functions
ERROR_MSG_VALUE_TOO_LOW	114	Raised when message value is insufficient for its correct running
ERROR_TOKEN_ACCOUNT_ALREADY_EXISTS	115	Raised upon an attempt to recreate token account that already exists
ERROR_INVALID_ROOT_CALLER	116	Raised when somebody but token root caller tries to invoke callback message
ERROR_ACCOUNT_DOES_NOT_EXIST	118	Raised upon an attempt to deposit token to unknown token account
ERROR_DEPOSIT_NOT_FOUND	119	Raised when the deposit to be topped up does not exist
ERROR_CALLER_IS_NOT_DEPOOL	120	Raised when transfer of values come from unregistered DePool
ERROR_DEPOSIT_WITH_SUCH_ID_EXISTS	121	Raised when trying to create a deposit with the same identifier as existing one
ERROR_PENDING_DEPOSIT_ALREADY_EXISTS	122	Raised when a deposit waiting for approval already exists as, according to the architecture more than one pending deposits is not allowed
ERROR_INVALID_DEPLOYER	123	Raised when somebody other than deployer tries to deploy a *Padawan*
ERROR_NOT_AUTHORIZED_VOTER	302	Raised when unauthorized voter tried to vote
ERROR_VOTING_NOT_STARTED	251	Raised when somebody tried to vote when voting is not started yet
ERROR_VOTING_HAS_ENDED	252	Raised when somebody tried to vote when voting is already

		completed
ERROR_VOTER_IS_NOT_ELIGIBLE	253	Raised when the voter is not white listed for the certain proposal

Fees

Name	Value	Description
START_BALANCE	3 ton	Initial balance of newly created *Padawan*
DEPLOY_FEE	3.1 ton	Minimal value required to deploy a new *Padawan*
DEPLOY_PROPOSAL_FEE	5 ton	Minimal value required to deploy a new *Proposal*
DEPOSIT_TONS_FEE	1 ton	Minimal value required to reclaim a deposit or to deposit TONs
DEPOSIT_TOKENS_FEE	1.5 ton	Minimal value required to deposit tokens
TOKEN_ACCOUNT_FEE	2 ton	Minimal value required to create a new token account
QUERY_STATUS_FEE	0.02 ton	Value sent to the *Proposal* to query status
DEF_RESPONSE_VALUE	0.03 ton	Value sent to the *Demiurge onProposalDeploy* message
DEF_COMPUTE_VALUE	0.02 ton	Value sent to the *Proposal finalize* and *initProposal*

Other

Name	Value	Description
PROPOSAL_HAS_WHITELIST	2	Bitmask that indicates that the *Proposal* accepts the restricted list of voters only

PROPOSAL_VOTE_SOFT_MAJORITY	4	Bitmask that indicates that soft majority voting model was chosen for the certain *Proposal*
PROPOSAL_VOTE_SUPER_MAJORITY	8	Bitmask that indicates that super majority voting model was chosen for the certain *Proposal*

Types

contractType

Enumerator that supports the contract kinds whose images are to be returned by the callback message. The following values are applicable:

- *Proposal*
- *Padawan*
- *PriceProvider*

Note!!! Much more values are defined but for the purposes of the present SMV specification, whereas just three mentioned above are applicable.

ProposalState

States for the current state of the proposal that can take the following values:

- *New* - the proposal has just been created
- *OnVoting* - the voting is ongoing
- *Ended* - the voting has been ended but the results are still unknown
- *Passed* - the voting has passed
- *Failed* - the voting has failed

Note!!! Much more values are defined, but for the purposes of the present SMV specification, just the five mentioned above are applicable.

VoteCountModel

An enumerator that represents the different kinds of voting models. It can accept the following values:

- *Majority* - the proposal passes in case “votes for” is greater than “votes against”
- *SoftMajority* - the proposal passes in case in satisfies the conditions described in corresponding section
- *SuperMajority* - the proposal passes in case in satisfies the conditions described in corresponding section

Note!!! Much more values are defined, but for the purposes of the present SMV specification, just the three mentioned above are applicable.

Deposit

This structure represents the deposit sent to the Padawan and has the following fields:

- *tokenID* - `uint256` value that can take the following values:
    - *0* - if it's a TON deposit
    - *1* - if it's a DePool deposit
    - *any other value* - address of the *ITokenRoot* (see below) contract casted to `uint256`
- *returnTo* - address of the deposit's owner where it has to be returned to
- *amount* - amount of deposit in TONs or tokens
- *approved*- a logical field that indicates if the deposit was approved or not. Applicable for token deposits only, for all the other types the value is always *true*. For token deposits, only one deposit for the specific Padawan can have this value as *false* (i.e. be pending)
- *depool* - applicable for DePool deposits only and states for the DePool address casted to `uint256`

ProposalInfo

This structure provides constant information about the Proposal. The following fields are defined:

- *id* - unique identifier of the proposal
- *start* - timestamp referring to the voting start
- *end* - timestamp referring to the voting end
- *options* - bitmask (`uint16`) that supports the following bit flags:
    - *PROPOSAL_HAS_WHITELIST* - only the eligible voters can vote
    - *PROPOSAL_VOTE_SOFT_MAJORITY* - soft majority voting count model must be used (the default one is majority)
    - *PROPOSAL_VOTE_SUPER_MAJORITY* - super majority voting count model must be used (the default one is majority)
    - Note that additional bit flags may be used for BFTG but the present SMV contract just uses three of them described above
- *totalVotes* - maximum amount of votes accepted
- *description* - string that describes the proposal
- *text* - string that describes the title of the proposal
- *voters* - the array of addresses of eligible voters. Used only if *PROPOSAL_HAS_WHITELIST* flag is on
- *ts* - timestamp that represents the time of proposal's creation
- *customData* - the TVM cell that represents the custom data such as *.pdf* document describing the proposal

ProposalData

This structure refers to the state of the proposal as well as to some supporting data. The fields of the structure are described below:

- *id* - unique identifier of the proposal
- *state* - state of the proposal represented by the ProposalState structure described above
- *userWalletAddress* - address of the wallet whose owner has submitted the proposal
- *addr* - address of the *Proposal* contract
- *ts* - timestamp that represents the time of proposal's creation
- *contestId* - not used for the purposes of the current SMV specification

VotingResults

This structure represents the final results of voting and has the following fields:

- *id* - unique identifier of the proposal
- *passed* - the logical field that indicates if the proposal passed or not
- *votesFor* - number of votes provided for the proposal
- *votesAgainst* - number of votes provided against the proposal
- *totalVotes* - total number of votes that can be accepted for (or against) the proposal
- *model* - vote counting model used for voting and represented by the VoteCountModel enumerator described above
- *ts* - timestamp that represents the time of creation of voting results

External contracts

DemiurgeStore

- *queryImage* - performs *updateImage* invocation as a callback providing an image for Padawan or Proposal contract depending on the parameters:
    - *kind* - contract type kind described by *ContractType* enumeration explained above
    - Invokes an *updateImage* callback for the *IDemiurgeStoreCallback* interface described below
- *queryDepools* - performs *updateDepools* invocation as a callback providing a collection of allowed DePools:
    - parameters - none
    - Invokes an *updateDepools* callback for the *IDemiurgeStoreCallback* interface described below
- *queryAddress* - performs *updateAddress* invocation as a callback providing an *PriceProvider* contract depending on the parameters:
    - *kind* - must be *PriceProvider*
    - Invokes an *updateAddress* callback for the *IDemiurgeStoreCallback* interface described below

Interfaces

IDemiurgeStoreCallback

This interface is required for the communication from DemiurgeStore to Demiurge (the latter implements the interface being discussed) and declares the following methods:

- *updateDepools(mapping(address => bool) depools)* - sets the collection of allowed depools together with the enabled status
- *updateImage(ContractType kind, TvmCell image)* - sets the image of the specific contract (*Padawan* or *Proposal*)
- *updateAddress(ContractType kind, address addr)* - sets the price provider (*kind* must be *PriceProvider*)

IDePool

This interface must be implemented by the DePool contract and provides the method for transferring stake to the *Padawan*. The interface declares the following methods:

- *transferStake(address dest, uint64 amount)* - tries to transfer the requested amount to the dest.

IPadawan

This interface must be implemented by the *Padawan* contract and must implement the following methods:

- *voteFor(address proposal, bool choice, uint32 votes)* - votes for (or against, depending on choice) the proposal with the certain number of votes
- *depositTons(uint32 tons)* - tries to deposit tons number of TONs
- *depositTokens(address returnTo, uint256 tokenId, uint64 tokens)* - tries to deposit tokens number of tokens identified by token root addressed at tokenId (casted to `uint256`) and to be returned to returnTo
- *reclaimDeposit(uint32 deposit)* - tries to reclaim deposit identified by deposit
- *confirmVote(uint64 pid, uint32 deposit)* - confirms votes for the *Proposal* identified by pid using deposit
- *rejectVote(uint64 pid, uint32 deposit, uint16 ec)* - rejects votes for the *Proposal* identified by pid using deposit providing the error code as ec
- *updateStatus(uint64 pid, ProposalState state)* - updates status for the *Proposal* identified by pid with state
- *createTokenAccount(address tokenRoot)* - creates new token account identified by tokenRoot
- *onTransfer(address source, uint128 amount)* - callback invoked by *DePool* contract mentioning its source and amount
- *getVoteInfo()* - returns the values of the *Padawan* voting variables - `_requestedVotes`, `_totalVotes` and LOCKED VOTES (as described at the section below)
- *getActiveProposals()* - returns the `_activeProposals`

IProposal

This interface must be implemented by the *Proposal* contract and must declare the following methods:

- *voteFor(address proposal, bool choice, uint32 votes)* - votes for (or against, depending on choice) the proposal with the certain number of votes
- queryStatus() - calls the *updateStatus* for the sender (who is supposed to be a Padawan)
- *wrapUp()* - tries to complete the voting

IClient

This optional interface may be implemented by the wallet. If so it's supposed to get an information using the declared methods:

- *updatePadawan(address addr)* - invoked when a *Padawan* with addr has been deployed
- *onProposalDeployed(uint32 id, address addr)* - callback infoking when external people deployed a new *Proposal*
- *onProposalCompletion(uint32 id, bool result)* - callback that is invoked when the voting is completed for the Proposal where *id* is a unique identifier of the *Proposal* and the result indicates if the *Proposal* passed

ITokenRoot

This interface must be implemented by each token root with the following methods:

- *deployEmptyWallet(uint32 _answer_id, int8 workchain_id, uint256 pubkey, uint256 internal_owner, uint128 grams)* - deploys a new token wallet with _answer_id callback (*onTokenWalletDeploy*), located at the workchain_id with pubkey (may be zero), internal_owner and supportive value of grams

ITokenWallet

This interface must be implemented by each token wallet with the following methods:

- *getBalance_InternalOwner(uint32 _answer_id)* - invokes the callback function identified by _answer_id with the parameter equals to the wallet balance
- *transfer(address dest, uint128 tokens, uint128 grams)* - transfers tokens to dest with supportive grams

IPriceProvider

This interface provides the current rate of TONs or tokens per vote. The following methods must be implemented:

- *queryTonsPerVote(uint32 queryId)* - sends the *updateTonsPerVote(uint32 queryId, uint64 price)* callback to the sender

- *queryTipsPerVote(uint32 queryId, address tokenRoot)* - sends the *updateTipsPerVote(uint32 queryId, uint64 price)* to the sender

Special variables

The following special “variables” are being used later throughout the specification.

- *caller* - message sender or zero in case of external message
- *value* - message value in TONs bound to the message
- *public key* - public key of the message sender
- *TVM public key* - public key of the initial contract creator
- *now* - returns the current Unix time as an Integer

Soft majority and supermajority voting models

This section is a discussion about the correct formulae for soft majority and supermajority vote counting model. Below it’s demonstrated that the formulae defined in the original specification can be incorrect and the alternatives are proposed.

Relations used to determine soft (super) majority success/failure voting

The following calculations were used:
for soft majority (1)

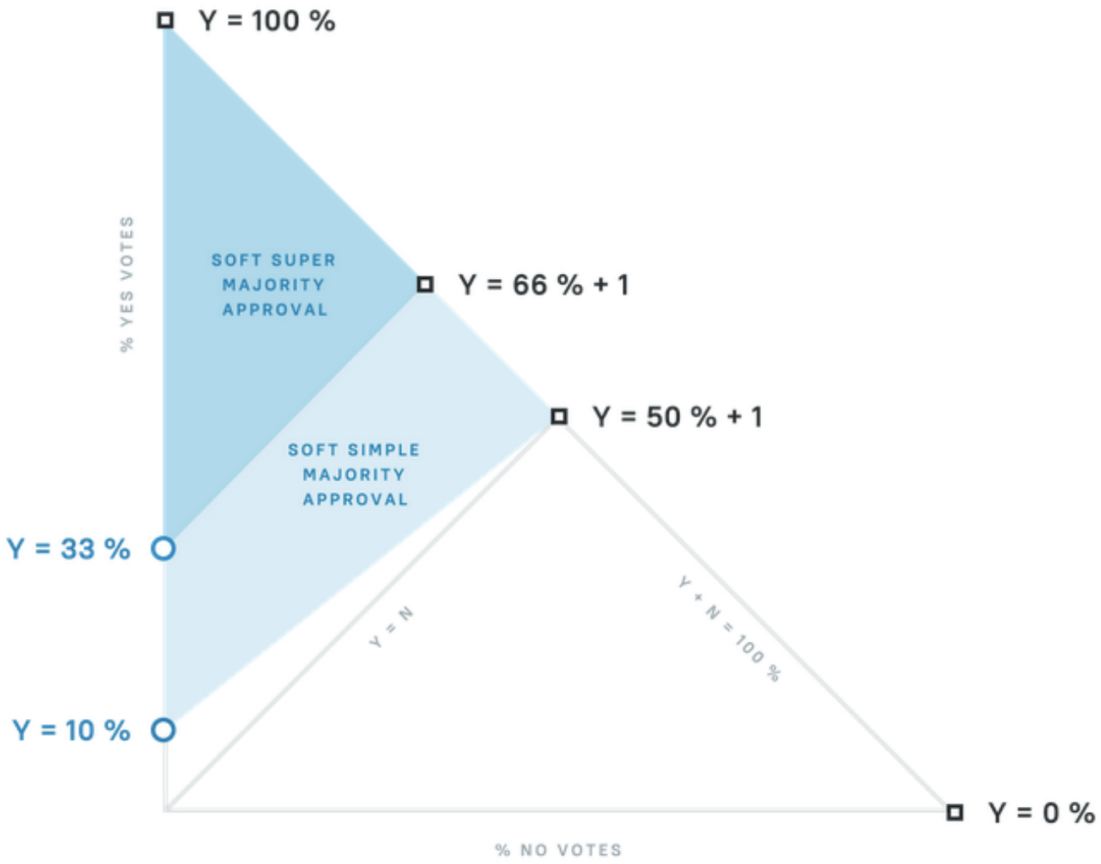
```
passed = (yes * total * 10 >= total * total + no * (8 * total  + 20));
```

for super majority (2)

```
passed = (yes * total * 3 >= total * total + no * (total + 6));
```

The explanations of these formulas could be found here
https://github.com/tonlabs/ton-labs-contracts/blob/master/governance/SMV/ProposalRoot.cpp
(lines 205-232).

The original graphic representations could be found here
https://firebasestorage.googleapis.com/v0/b/ton-labs.appspot.com/o/documents%2Fapplication%2Fpdf%2Fiwl3spkopkkjoxt067-RSquad_SMV_final_ENG_revised.pdf?alt=media&token=343d1570-b28f-470b-8552-1cade1ebfad9 (page 19).



We place the copy of the image for reader's comfort.

However, simple tests show some inconsistencies with the formulas.

Take, for example values, t = 21, y = 11, n = 10. According to common sense (one can find a lot of explanations, see e.g. https://en.wikipedia.org/wiki/Majority) the majority should work in this case as 50% of 21 should be rounded down to 10. However, using (1) we have

passed = (11*21*10 > 21*21+10*(8*21+20)) = 2310 > 441+1880 = 2310 > 2321 = false, instead of true.

The origin of this error lies in operations of multiplication by the denominator, which usually is correct but not when using rounded integer numbers so [a/b]*b ≠ a.

To fix this issue we propose the following algorithm to check whether the voting is passed:
(1') passed = y>=1+[t/10]+[(n*([t/2]-[t/10]))/[t/2]] where [] denotes rounded down (floor) value of division.
One can easily get an analogous (2') formula:
(2') passed = y>=1+[t/3]+[(n*([2*t/3]-[t/3]))/[t/3]]

The following tables show some supporting calculations to realize the correctness (the full proof can be done at the next stages of verification).

t	n	y >=	passed
3	0	1	TRUE
	1	2	TRUE
	2	3	FALSE
	3	4	FALSE
10	0	2	TRUE
	1	2	TRUE
	2	3	TRUE
	3	4	TRUE
	4	5	TRUE
	5	6	FALSE
	6	6	FALSE
	7	7	FALSE
	8	8	FALSE
	9	9	FALSE
	10	10	FALSE

t	n	y >=	passed
11	0	2	TRUE
	1	2	TRUE
	2	3	TRUE
	3	4	TRUE
	4	5	TRUE
	5	6	TRUE
	6	6	FALSE
	7	7	FALSE
	8	8	FALSE
	9	9	FALSE

	10	10	FALSE
	11	10	FALSE
21	0	3	TRUE
	1	3	TRUE
	2	4	TRUE
	3	5	TRUE
	4	6	TRUE
	5	7	TRUE
	6	7	TRUE
	7	8	TRUE
	8	9	TRUE
	9	10	TRUE
	10	11	TRUE
	11	11	FALSE
	12	12	FALSE
	13	13	FALSE
	14	14	FALSE
	15	15	FALSE
	16	15	FALSE
	17	16	FALSE
	18	17	FALSE
	19	18	FALSE
	20	19	FALSE
	21	19	FALSE

Demiurge smart contract

State

The full state of the Demiurge contract may be described by the table below. It's important to mention that this state is using the external observer's point of view (some kind of God mode)

so some elements may not be kept by the real implementation. For example, the real implementation may not keep the list of the token contracts, but for an external observer, a list like this is still a part of the state.

Variable	Subvariable	Subsubvariable	Description
`_padawanSI`			*TVMCell* variable that describes the Padawan state image. MAY BE ZERO.
`_proposalSI`			*TVMCell* variable that describes the Proposal state image. MAY BE ZERO.
`_priceProvider`			Address of PriceProvider contract. MAY BE ZERO. MUST IMPLEMENT IPriceProvider
`_deployedPadawans`			The collection of deployed Padawans
	`padawanId`		Id of the Padawan. MUST BE UNIQUE
	`userWalletAddress`		The wallet of the Padawan's owner. MUST NOT BE ZERO
	`addr`		Address of the Padawan. MUST NOT BE ZERO.
`_deployedProposals`			The collection of deployed Proposals
	`address`		Address of the Proposal. MUST NOT BE ZERO
	`id`		Id of the Proposal. MUST BE UNIQUE
	`proposalInfo`		ProposalInfo structure
		`id`	Proposal id. MUST BE

			EQUAL TO *id* OF *proposal*.
		`start`	Voting start (as an *uint32* timestamp). MUST BE POSITIVE
		`end`	Voting end (as an *uint32* timestamp). MUST BE POSITIVE
		`options`	bit mask that may contain *PROPOSAL_VOTE_SOFT_MAJORITY*, *PROPOSAL_VOTE_SUPER_MAJORITY*, *PROPOSAL_HAS_WHITELIST*
		`totalVotes`	The maximum number of votes that can be granted to proposal
		`description`	Proposal description
		`text`	Proposal title
		`voters`	List of eligible voters. MAY BE EMPTY
		`ts`	Creation time (as an *uint32* timestamp). MUST BE POSITIVE
		`customData`	TVMCell that contains custom data such as *.pdf* file
	`proposalData`		*ProposalData* structure
		`id`	Proposal id. MUST BE EQUAL TO *id* OF *proposal*.
		`ProposalState`	State of the proposal. MUST BE EITHER *New*, *OnVoting*, *Ended*, *Passed* or *Failed*

		`userWalletAddress`	Address of the creator
		`addr`	Address of the Proposal. MUST BE EQUAL TO *addr* OF *proposal*
		`ts`	Creation time. MUST BE EQUAL TO `ts` OF *proposalInfo*
		`contestId`	Not used
`depools`			The collection of allowed DePools
	`address`		Address of DePool contract. MUST IMPLEMENT *IDePool* interface
	`enabled`		Boolean variable that indicates if DePool enabled or not
`demiStore`			Address of the external contract that MUST IMPLEMENT *DemiurgeStore* interface. MAY BE ZERO. This external contract contains all the images of auxiliary contracts such as Padawan or Proposal
`_votingResults`			Historical collection of passed proposals
	`id`		*id* OF *proposal*
	`passed`		boolean parameter that indicates if the voting passed
	`votesFor`		Number of “yes” votes
	`votesAgainst`		Number of “no” votes

	`totalVotes`		*totalVotes* OF *proposalInfo* OF *proposal*
	`model`		Voting model as *VotingCountModel*
	`ts`		Time of finalizing the results (as *uint32* timestamp)
`CHECK_PADAWAN`	`boolean`		true if Padawan image is set, false otherwise
`CHECK_PROPOSAL`	`boolean`		true if Proposal image is set, false otherwise
`CHECK_DEPOOLS`	`boolean`		true if collection of DePools is set, false otherwise
`CHECK_PRICE_PROVIDER`	`boolean`		true if `_priceProvider` is set, false otherwise

Special Stories

checkPassed

This story is about ensuring that all the required images and addresses were set up.

The formalized statements here are:

- CHECK PASSED - `CHECK_PADAWAN` AND `CHECK_PROPOSAL` AND `CHECK_DEPOOLS` AND `CHECK_PRICE_PROVIDER`

buildStateInit

This story is about the conversion of an image represented by TVMCell into another TVMCell that can be used for contact initialization.

Formalized language here:

- STATE FROM *image* FOR *name* WITH *key* - the result of subsequent call of the following operations:

- *image* converted to slice[8]
- first reference[9] is taken from the resulting slice
- *tvm.buildStateInit* is called with the following arguments:
  - *contr* - name
  - *varInit* - the following list of variables:
    - *deployer* - address of the Demiurge contract itself
  - *pubkey* - key
  - *code* - the reference received at the point above
- The resulting *TVMCell* is the outcome of the operation

deployedPadawans

This story is about the interpretation of the `_deployedPadawans`.

The following statements are introduced here:

- PADAWAN ADDRESSED *addr* - *padawan* WHERE *addr* OF *padawan* EQUALS TO *addr* AND *padawan* IN `_deployedPadawans`
- PADAWAN IDED *id* - *padawan* WHERE *padawanId* OF *padawan* EQUALS TO *id* AND *padawan* IN `_deployedPadawans`

deployedProposals

This story is about the interpretation of the `_deployedProposals`.

The following statements are introduced here:

- PROPOSAL ADDRESSED *addr* - *proposal* WHERE *addr* OF *proposal* EQUALS TO *addr* AND *proposal* IN `_deployedProposals`

deployProposal

This story is about deploying a new proposal.

The following statements are used:

- DEPLOY PROPOSAL WITH *totalVotes*, *start*, *end*, *options*, *specificData*, *description*, *text*, *model*, *voters*, *id* - is a subsequent invocation of the following activities:
  - IF *model* IS *VoteCountModel.SoftMajority* THEN

[8] A TVM cell slice, or slice for short, is a contiguous “sub-cell” of an existing cell, containing some of its bits of data and some of its references. Essentially, a slice is a read-only view for a subcell of a cell. Slices are used for unpacking data previously stored (or serialized) in a cell or a tree of cells.
[9] Each slice has up to four references. The first one is taken. If the image is broken and does not have any references the exception is raised

        - *real options* IS LOGICAL OR OF *options* AND *PROPOSAL_VOTE_SOFT_MAJORITY*
    - IF *model* IS *VoteCountModel.SuperMajority* THEN
        - *real options* IS LOGICAL OR OF *options* AND *PROPOSAL_VOTE_SUPER_MAJORITY*
    - OTHERWISE *real options* IS *options*
    - new proposal IS CREATED with the following parameters:
        - *state* IS STATE FROM `_proposalSI` FOR 'Proposal' WITH *id*
        - *value* IS *START_BALANCE*
        - *proposalInfo* IS :
            - *id* - *id*
            - *start* - *start*
            - *end* - *end*
            - *options* - *real options*
            - *totalVotes* - *totalVotes*
            - *description* - *description*
            - *text* - *text*
            - *voters* - *voters*
            - *ts* - *now*
            - *specificData* - *specificData*
        - *proposalData* IS :
            - *id* - *id*
            - *ProposalState* - *ProposalState.New*
            - userWalletAddr - caller
            - addr - ADDRESS OF *new proposal*
            - *ts* - *now*
            - *contestId* - NOT SPECIFIED
    - new proposal MUST BE ADDED TO _deployedProposals WITH:
        - *address* - ADDRESS OF *new proposal*
        - *id* - *id*

Demiurge Contract Functions: A Specification

Important note!!! Some implementations may have additional parameters for some functions described above. It may be necessary for some extra functionality supported by such implementations (such as BFTG), but for the present specification such extras are ignored.

constructor(address store)

The main role of the constructor is to ask IDemiurgeStore for all the images. However, it's possible not to set the store and later fill all the required fields, manually using the corresponding methods.

Parameter Name	Parameter Type	Description
`store`	`address`	Address of the `tokenRoot` to be used by the Padawan. MAY BE ZERO. IF NOT ZERO THAN MUST IMPLEMENT ITokenRoot interface

Function Access Restrictions

1. *caller* MUST BE EITHER INTERNAL OR *public key* MUST BE EQUAL TO *TVM public key* OTHERWISE 101

Outcome

1. `demiStore` IS store
2. `_padawanSI` IS ZERO
3. `_proposalSI` IS ZERO
4. `_priceProvider` IS ZERO
5. `_deployedPadawans` IS EMPTY
6. `_deployedProposals` IS EMPTY
7. `depools` IS EMPTY
8. `_votingResults` IS EMPTY
9. `CHECK_PADAWAN` IS FALSE
10. `CHECK_PROPOSAL` IS FALSE
11. `CHECK_DEPOOLS` IS FALSE
12. `CHECK_PRICE_PROVIDER` IS FALSE
13. Gas MUST BE accepted after requires

Messages

1. IF store IS NOT ZERO THEN:
    a. *queryImage* message MUST BE SENT TO store with the following parameters:
        i. *value* - 0.2 ton
        ii. *kind - ContractType.Padawan*
    b. *queryImage* message MUST BE SENT TO store with the following parameters:
        i. *value* - 0.2 ton
        ii. *kind - ContractType.Proposal*
    c. *queryAddress* message MUST BE SENT TO store with the following parameters:
        i. *value* - 0.2 ton
        ii. *kind - ContractType.PriceProvider*
    d. *queryDepools* message MUST BE SENT TO store with the following parameters:
        i. *value* - 0.2 ton

Errors

1. Nothing but mentioned above

updateImage(ContractType kind, TvmCell image)

It's a callback invoked by *DemiurgeStore* (or as an external message if *DemiurgeStore* is not set) and sets images for Padawans and Proposals.

Parameter Name	Parameter Type	Description
`kind`	`ContractType`	Kind of the contract whose image is updated
`image`	`TvmCell`	MUST BE A VALID IMAGE OF THE CONTRACT

Function Access Restrictions

1. *caller* MUST BE `demiStore` OTHERWISE EXCEPTION

Outcome

1. IF kind IS *ContractType.Padawan* THEN
    a. `CHECK_PADAWAN` MUST BE TRUE
    b. `_padawanSI` MUST BE image
2. IF kind IS *ContractType.Proposal* THEN
    a. `CHECK_PROPOSAL` MUST BE TRUE
    b. `_proposalSI` MUST BE image
3. OTHERWISE
    a. State is not changed

Errors

1. Nothing but mentioned above

updateDepools(mapping(address => bool) depools)

It's a callback invoked by *DemiurgeStore* (or as an external message if *DemiurgeStore* is not set) and sets `depools` collection.

Parameter Name	Parameter Type	Description
`depools`	`mapping (address => bool)`	The collection of depools

Function Access Restrictions

1. *caller* MUST BE `demiStore` OTHERWISE EXCEPTION

Outcome

1. `CHECK_DEPOOLS` MUST BE TRUE
2. `depools` MUST BE depools

Errors

1. Nothing but mentioned above

updateAddress(ContractType kind, address addr)

It's a callback invoked by *DemiurgeStore* (or as an external message if *DemiurgeStore* is not set) and sets the address for `_priceProvider`.

Parameter Name	Parameter Type	Description
`kind`	`ContractType`	Kind of the contract whose image is updated
`addr`	`address`	MUST BE NOT ZERO. MUST IMPLEMENT *IPriceProvider* interface

Function Access Restrictions

1. *caller* MUST BE `demiStore` OTHERWISE EXCEPTION

Outcome

1. IF kind IS *ContractType.PriceProvider* THEN
    b. `CHECK_PRICE_PROVIDER` MUST BE TRUE
    c. `_priceProvider` MUST BE addr
2. OTHERWISE

a. State is not changed

Errors

1. Nothing but mentioned above

deployPadawan(uint userKey)

This function deploys a new Padawan and adds it to the collection of `_deployedPadawans`. However it does not initialize it (as it's being done by the callback received from the newly created Padawan).

Parameter Name	Parameter Type	Description
`userKey`	`integer`	Public key of the newly created Padawan

Function Access Restrictions

1. No restrictions

Outcome

1. *new padawan* IS CREATED with the following arguments:
   a. *state* - STATE FROM `_padawanSI` FOR 'Padawan' WITH userKey
   b. *value* - *START_BALANCE*
2. *new padawan* MUST BE ADDED to _deployedPadawans with the following parameters:
   a. *padawanId* IS userKey
   b. *userWalletAddress* IS *caller*
   c. addr - ADDRESS OF *new padawan*

Messages

1. constructor is called for new padawan

Errors

1. IF *padawan* WITH padawanId OF *padawan* EQUALS TO userKey EXISTS IN `_deployedPadawans` THEN *ERROR_PADAWAN_ALREADY_DEPLOYED*
2. IF *value* IS LESS THAN *DEPLOY_FEE* THAN EXCEPTION

onPadawanDeploy(uint key)

This is a callback function that is invoked by a newly deployed Padawan contract upon its

creation. Validates the Padawan, sends the *initPadawan* message to the Padawan as well as an *updatePadawan* to its wallet.

Parameter Name	Parameter Type	Description
`key`	`integer`	Public key of the newly created Padawan

Function Access Restrictions

1. *caller* MUST BE *addr* OF PADAWAN ADDRESSED key

Outcome

1. State is not changed

Messages

1. *initPadawan* MESSAGE MUST BE SENT TO *addr* OF PADAWAN ADDRESSED key with the following parameters:
    a. *value* MUST BE ZERO
    b. *wallet* MUST BE *userWalletAddress* OF PADAWAN ADDRESSED key
    c. *voteProvider* MUST BE `_priceProvider`
    d. *depoolAddrs* MUST BE `depools`
2. *updatePadawan* (supposing that the recipient supports *IClient* interface) MUST BE SENT TO *userWalletAddress* OF PADAWAN ADDRESSED key with the following parameters:
    a. *addr* - *addr* OF PADAWAN ADDRESSED key

Errors

1. Nothing but mentioned above

deployProposal(uint32 totalVotes, uint32 start, uint32 end, string description, string text, VoteCountModel model)

This function deploys an open proposal where all the `_deployedPadawan` can vote. It requires totalVotes to pass, starts at start, ends at the end, titled by text, described by description and uses a model for counting.

Parameter Name	Parameter Type	Description

`totalVotes`	`integer (uint32)`	Votes required for Proposal to pass
`start`	`integer (uint32)`	Timestamp for voting start
`end`	`integer (uint32)`	Timestamp for voting end
`description`	`string`	The detailed description of the proposal
`text`	`string`	The title of the proposal
`model`	`VoteCountModel`	MUST BE EITHER *VoteCountModel.Majority* OR *VoteCountModel.SoftMajority* OR *VoteCountModel.SuperMajority*

Function Access Restrictions

1. *caller* MUST BE *addr* OF PADAWAN ADDRESSED key

Outcome

1. DEPLOY_PROPOSAL_WITH totalVotes, start, end, *DEFAULT_OPTIONS*, EMPTY CELL, description, text, model, EMPTY, UNIQUE KEY

Messages

1. Constructor for *new proposal* is called

Errors

1. IF value IS LESS THAN *DEPLOY_PROPOSAL_FEE* THEN EXCEPTION

deployProposalWithWhitelist(uint32 totalVotes, uint32 start, uint32 end, string description, string text, VoteCountModel model, address[] voters)

This function deploys a restricted proposal where only some “whitelisted” members of the `_deployedPadawan` can vote. It requires totalVotes to pass, starts at start, ends at the end, titled by text, described by description and uses a model for counting.

Parameter Name	Parameter Type	Description
`totalVotes`	`integer (uint32)`	Votes required for Proposal to pass
`start`	`integer (uint32)`	Timestamp for voting start
`end`	`integer (uint32)`	Timestamp for voting end
`description`	`string`	The detailed description of the proposal
`text`	`string`	The title of the proposal
`model`	`VoteCountModel`	MUST BE EITHER *VoteCountModel.SoftMajority* OR *VoteCountModel.SuperMajority*
`voters`	`address[]`	The list of eligible voters

Function Access Restrictions

1. *caller* MUST BE *addr* OF PADAWAN ADDRESSED key

Outcome

1. DEPLOY_PROPOSAL_WITH totalVotes, start, end, *PROPOSAL_HAS_WHITELIST*, EMPTY CELL, description, text, model, voters, UNIQUE KEY

Messages

1. Constructor for *new proposal* is called

Errors

1. IF value IS LESS THAN *DEPLOY_PROPOSAL_FEE* THEN EXCEPTION

onProposalDeploy()

This function is a callback that is invoked by *proposal* IN `_deployedProposals` to complete the *proposal* initialization. In their turn it invokes *initProposal* callback back to the *proposal* as well as an optional *onProposalDeployed* one to the `userWalletAddress` in case it implements *IClient* interface.

Function Access Restrictions

1. *caller* MUST BE IN `_deployedProposals` OTHERWISE EXCEPTION

Outcome

1. State is not changed

Messages

1. *initProposal* message MUST BE SENT TO PROPOSAL ADDRESSED *caller* with the following parameters:
    a. value MUST BE EQUAL TO DEF_COMPUTE_VALUE
    b. pi MUST BE EQUAL TO *proposalInfo* OF PROPOSAL ADDRESSED *caller*
    c. padawanSI MUST BE EQUAL TO `_padawanSI`
2. *onProposalDeployed* MUST BE SENT TO *userWalletAddress* OF *proposalInfo* OF PROPOSAL ADDRESSED *caller* with the following parameters (note : the implementation of IClient interface is optional for the receiver):
    a. *id* - *id* OF PROPOSAL ADDRESSED *caller*
    b. *addr* - *addr* OF PROPOSAL ADDRESSED *caller*

Errors

1. Nothing but mentioned above

onStateUpdate(ProposalState state)

This function is a callback sent by *proposal* IN `_deployedProposals` when the state of *proposal* has been changed. Changes the internal state of the *proposal*.

Parameter Name	Parameter Type	Description
`state`	`ProposalState`	New *ProposalState* for the *caller*

Function Access Restrictions

1. *caller* MUST BE IN `_deployedProposals` OTHERWISE EXCEPTION

Outcome

1. *ProposalState* OF *proposalData* OF PROPOSAL ADDRESSED *caller* MUST BE state
2. Change MUST BE RETURNED

Errors

1. Nothing but mentioned above

reportResults(VotingResults results)

This function is a callback sent by *proposal* IN `_deployedProposals` when the voting is passed and results are ready. It keeps the results in `_votingResults` for historical purposes and sends a message *onProposalCompletion* message to the *userDataWallet* OF *proposalData* OF *proposal* (optional, in case it implements *IClient* interface).

Parameter Name	Parameter Type	Description
`results`	`VotingResults`	Results of the voting sent by the *caller*

Function Access Restrictions

1. *caller* MUST BE IN `_deployedProposals` OTHERWISE EXCEPTION

Outcome

1. results ARE ADDED TO `_votingResults`

Messages

1. *onProposalCompletion* message MUST BE SENT TO *userWalletAddress* OF *proposalInfo* OF PROPOSAL ADDRESSED *caller* with the following parameters (note : the implementation of IClient interface is optional for the receiver):
    a. *value* - *DEF_COMPUTE_VALUE*
    b. id - *id* OF PROPOSAL ADDRESSED *caller*
    c. *result* - *passed* OF results

setProposalSI(TvmCell c)

Sets the Proposal image in case it was not before.

Parameter Name	Parameter Type	Description
c	TVMCell	Image of the Proposal contract

Function Access Restrictions

1. No restrictions

Outcome

1. _proposalSI MUST BE EQUAL TO c
2. CHECK_PROPOSAL MUST BE TRUE

Errors

1. IF CHECK_PROPOSAL IS TRUE THEN EXCEPTION

setPadawanSI(TvmCell c)

Sets the Padawan image in case it was not before.

Parameter Name	Parameter Type	Description
c	TVMCell	Image of the Padawan contract

Function Access Restrictions

1. No restrictions

Outcome

1. _padawanSI MUST BE EQUAL TO c
2. CHECK_PADAWAN MUST BE TRUE

Errors

1. IF CHECK_PADAWAN IS TRUE THEN EXCEPTION

setPriceProvider(address addr)

Sets the _priceProvider in case it was not before.

Parameter Name	Parameter Type	Description
`addr`	`address`	Address of price provider. MUST IMPLEMENT *IPriceProvider* INTERFACE. MUST NOT BE ZERO

Function Access Restrictions

1. No restrictions

Outcome

1. `_priceProvider` MUST BE EQUAL TO addr
2. `CHECK_PRICE_PROVIDER` MUST BE TRUE

Errors

1. IF `CHECK_PRICE_PROVIDER` IS TRUE THEN EXCEPTION

setDePool(address addr)

Adds addr to the `depools` in case set up is not completed yet, so CHECK PASSED IS FALSE.

Parameter Name	Parameter Type	Description
`addr`	`address`	Address of DePool. MUST IMPLEMENT *IDePool* INTERFACE. MUST NOT BE ZERO

Function Access Restrictions

1. No restrictions

Outcome

1. *new depool* MUST BE CREATED with the following parameters:
    a. *address* MUST BE EQUAL TO addr
    b. *enabled* MUST BE TRUE

2. *new depool* MUST BE ADDED TO depools
3. `CHECK_DEPOOLS` MUST BE TRUE

Errors

1. IF `CHECK PASSED` IS TRUE THEN EXCEPTION

getImages()

A public function that returns images of `_padawanSI` and `_proposalSI`.

Function Access Restrictions

1. No restrictions

Outcome

1. State is not changed
2. Return value is the pair of:
    a. `_padawanSI`
    b. `_proposalSI`

Errors

1. None

getPartners()

A public function that returns `_priceProvider` and the list of enabled `depools`.

Function Access Restrictions

1. No restrictions

Outcome

1. State is not changed
2. Return value is the pair of:
    a. `_priceProvider`
    b. ALL THE *address* OF *depool* WHERE *depool* IN `depools` AND *enabled* OF *depool* IS TRUE

Errors

1. None

getDeployed()

A public function that returns a collection of `_deployedPadawans` and `_deployedProposals`.

Function Access Restrictions

1. No restrictions

Outcome

1. State is not changed
2. Return value is the pair of:
    a. `_deployedPadawans`
    b. `_deployedProposals`

Errors

1. None

getVotingResults()

A public function that returns a collection of historical `_votingResults`.

Function Access Restrictions

1. No restrictions

Outcome

1. State is not changed
2. Return value IS `_votingResults`

Errors

1. None

getProposalInfo()

A public function that returns a collection of *proposalInfo* of all the `_deployedProposals`..

Function Access Restrictions

1. No restrictions

Outcome

1. State is not changed
2. Return value IS `_deployedProposals` MAPPED TO *proposalInfo*

Errors

1. None

getProposalData()

A public function that returns a collection of *proposalData* of all the `_deployedProposals`..

Function Access Restrictions

1. No restrictions

Outcome

1. State is not changed
2. Return value IS `_deployedProposals` MAPPED TO *proposalData*

Errors

1. None

getStats()

A public function that returns a generic information about the version of the system as well as about the number of padawans and proposals deployed.

Function Access Restrictions

1. No restrictions

Outcome

1. State is not changed
2. Return value is a tuple of:
    a. *version* MUST BE 2
    b. *deployedPadawansCounter* MUST BE A NUMBER OF `_deployedPadawans` (**discussable!! only those padawans that passed onPadawanDeploy message**)
    c. *deployedProposalsCounter* MUST BE A NUMBER OF `_deployedProposals` (**discussable!! only those padawans that passed onPadawanDeploy message**)

Errors

1. None

getPadawan(uint key)

This function returns a PADAWAN IDED key or an exception is raised.

Parameter Name	Parameter Type	Description
`key`	`uint`	*padawanID* of the `_deployedPadawans`

Function Access Restrictions

1. No restrictions

Outcome

1. State is not changed
2. Return value is the pair of:
    a. *userWalletAddress* - *userWalletAddress* OF PADAWAN IDED key
    b. *addr* - *addr* OF PADAWAN IDED key

Errors

1. IF NO PADAWAN IDED key THEN EXCEPTION

Padawan smart contract description

What is a Padawan smart contract?

A Padawan smart contract is a user ballot that allows users to vote for proposals. Padawan accepts deposits of different types (TONs, TIP3 tokens, DePool stakes), converts them into votes, and sends the votes to proposals. The votes cannot be converted into deposits and received back until all the proposals that the Padawan has voted for are completed.

A user can vote for different proposals with a different number of votes, but a number of locked votes in a Padawan is always the maximum number of votes spent for one proposal. At any time, a user can ask to reclaim some deposits equivalent to a number of votes. When this happens, the Padawan starts to query the status of all voted proposals. If any of them has already been completed, the Padawan removes it from the active proposals list and updates the value of locked votes. If the required number of votes becomes less or equal to unlocked votes, then the Padawan converts the requested number of votes into a deposit (TONs, TIP3 tokens or DePool stakes) and sends it back to the user.

A Padawan is controlled by a user contract that requested *deployPadawan* from Demiurge.

State of the Padawan contract

The full state of the Padawan contract may be described by the table below. It's important to mention that this state is using the external observer's point of view (some kind of God mode) so some elements may not be kept by the real implementation. For example, the real implementation may not keep the list of the token contracts but for the external observer such a list is still a part of the state.

Element		Description
`deployer`		Address of the deployer, supposed to be an instance of Demiurge
`_wallet`		Padawan owner address
`_priceProvider`		PriceProvider contract address
`tokenAccounts`		Collection of Padawan's token accounts
	tokenRoot	Token root of the wallet
	address	Address of the account

	walletKey	Address of the owner
	createdAt	Unique Id (Time of creation may be an option)
	balance	Balance of the token
`deposits`		Map of deposits of different currencies (TON Crystals, TIP3 tokens, DePool stake) where key is the root token address and the
	id	Deposit Id. MUST BE UNIQUE. *now* may be used as an option.
	tokenId	ZERO IN CASE it's a deposit in TONs. tokenId that is a STANDARD ADDRESS of the token root IN CASE it's a deposit in tokens. IN CASE it's a DePool deposit MUST BE EQUAL to ONE
	returnTo	Address of the wallet where the deposit must be returned to
	amount	Amount of deposits in TONs or tokens
	valuePerVote	Price per vote in TONs or tokens
	approved	Approved or pending
	depool	In case it's a DePool deposit - depool ID, otherwise ZERO
`_activeProposals`		Set of proposal addresses for which the user has voted and which are not finalized yet, described in a special story
`_requestedVotes`		Number of votes requested to

		be reclaimed
`_totalVotes`		Total number of votes available to the user
`_spentVotes`		Number of votes spent for each proposal
`_pendingDepositID`		ID of token deposit that is not approved yet, zero if there is no pending deposits
`depools`		Collection of addressed of DePool[10] contracts used by Padawan where each DePool can be either enabled or disabled

Special stories

_activeProposals

This substate keeps the number of voices granted for each proposal. The following formalized language is used for this story:

- VOICES FOR *proposal* - number of *votes* granted by the *present padawan* to the *proposal* identified by its address in `_activeProposals`
- INITIAL ACTIVE PROPOSALS - VOICES FOR EACH *proposal* ARE EQUAL TO ZERO
- VOTED PROPOSAL - *proposal* IN `_activeProposals` WHERE VOICES FOR *proposal* IS POSITIVE
- VOICES FOR *proposal* IS INCREASED BY *votes*
    - FOR ANY *another proposal* that is NOT EQUAL TO *proposal* VOICES FOR *another proposal* UNCHANGED

Additional requirements:

- VOICES FOR ANY *proposal* ARE NOT NEGATIVE
- VOICES FOR ANY *proposal* LESS OR EQUAL THAN `_totalVotes`

_spentVotes

This substate holds the number of active proposals to which the Padawan provided a certain number of votes for each number of votes. The maximum number of votes where spent votes

[10] https://docs.ton.dev/86757ecb2/p/45d6eb-depool-specifications

are positive can be considered as locked votes for padawan that must be held in case of revoking the deposit.
The following formalized language is used for this story:

- SPENT VOTES FOR *votes* - `_spentVotes` for *votes*
- LOCKED VOTES - THE MAXIMAL *votes* WHERE SPENT VOTES FOR *votes* IS POSITIVE
- INITIAL SPENT VOTES - SPENT VOTES FOR EACH *vote* ARE ZERO
- INCREASE SPENT VOTES FOR *votes* - SPENT VOTES FOR *votes* IS INCREASED BY ONE
- DECREASE SPENT VOTES FOR *votes* - SPENT VOTES FOR *votes* IS DECREASED BY ONE

Additional requirements:

- IF *votes* IS POSITIVE THAN SPENT VOTES FOR *votes* IS NOT NEGATIVE
- IF *votes* IS NOT POSITIVE THAN SPENT VOTES FOR *votes* IS ARBITRARY OR NOT DEFINED

tipAccount

A combined information about the token that has:

- Address
- Owner address
- Time of creation
- Balance

The following states are declared for this substate:

- INITIAL TIP for the owner. The result is the object with the following attributes:
  - Address - 0
  - Owner address - owner
  - Time of creation - NOW
  - Balance - 0
  - Side effect : the deploy Wallet MESSAGE MUST BE SENT TO _tokenRoot THAT MUST IMPLEMENT WITH VALUE {value: 2 ton, flag: 1, bounce: true} and the following parameters
    - answerId - CALLBACK ID (named *onTokenWalletDeploy*)
    - workchain_id - ZERO
    - pubkey - ZERO
    - internal_owner - deployer
    - grams - 1 ton
- TRANSFER TIPS FOR deposit WITH value
  - tipAccount - TIP WITH ADDRESS FROM tokenId of deposit FROM tokenAccounts_voters_voters
  - ITokenWallet transfer METHOD INVOKED FOR address of tipAccount WITH :

    - Value - 0.2 ton
    - Input parameters:
        - Dest - returnTo of deposit
        - Value - value
        - Grams - 0.1 ton

tokenAccounts

The collection of all the supported token accounts handled as a key-pair of token root and tipAccount.

The following statements are declared for this story:

- TIP IDED *tokenId* - tipAccount FROM `tokenAccounts` WITH *address* EQUAL TO STANDARD ADDRESS FROM *tokenId* OTHERWISE ERROR_ACCOUNT_DOES_NOT_EXIST
- TIP ADDRESSED *addr* - *tip* IN `tokenAccounts` AND *walletKey* OF *tip* IS *addr*

deposits

This story is about `deposits`. The following statements are used:

- DEPOSIT IDED *id* - *deposit* WHERE *id* OF *deposit* IS id AND *deposit* IN `deposits`
- PENDING DEPOSIT - DEPOSIT IDED `_pendingDepositId`

depools

This story describes the collection of depools where each depool may be available or not. The formalized language is as follows:

- INITIAL DEPOOLS - IS EMPTY

unlockDeposits

This story does the following : tries to reclaim amount corresponding to the value of _requestedVotes from the deposits starting from the bigger one until either _requestedVotes become zero or no more votes can be converted from deposits. The method affects _requestedVotes and deposits values of the state as well as sends either TON crystals (via transfer method) or tokens (via returnTo method of the deposit).

IMPORTANT note!!! The authors of the present specification intentionally omit the case when the deposit to be reclaimed tied to DePool. The reason is as follows: the architecture of the overall solution relies on the *transferStake* DePool method and considers it as reliable (so, the requested stake transfer is supposed to be guaranteed).

However, this method makes the transfer only when the stake is not included into the current Elector's deposit otherwise it transfers partially or is not transferred at all. The authors consider it as a critical issue of the current architecture and suppose the overall DePool process will be revisited at the architectural level. So at the moment this option is not included into the present specification.

Formalized language:

- RECLAIMABLE *deposit* - *deposit* WHERE amount OF *deposit* IS GREATER OR EQUAL THAN valuePerVote OF *deposit* AND *deposit* IN `deposits`
- TOTAL AMOUNT OF DEPOSITS - SUM OF amount of ALL *deposits* IN `deposits`
- RECLAIMED VALUE OF *deposit* - THE DIFFERENCE BETWEEN amount OF NEW *deposit* AND amount OF OLD *deposit*
    - RECLAIMED VALUE OF *deposit* MUST BE DIVISIBLE TO valuePerVote OF *deposit*
- RECLAIMED VOTES OF deposit - RECLAIMED VALUE OF deposit DIVIDED TO valuePerVote OF *deposit*
- TOTAL RECLAIMED VOTES - SUM OF RECLAIMED VOTES OF ALL deposits in `deposits`
- RECLAIMED *deposit* - RECLAIMED VALUE OF *deposit* IS POSITIVE
- RECLAIMED TON *deposit* - RECLAIMED *deposit* AND tokenId OF *deposit* IS ZERO
- RECLAIMED TOKEN deposit - RECLAIMED *deposit* AND tokenId OF *deposit* IS NEITHER ZERO NOR ONE
- VALUE OF RECLAIMED TON DEPOSITS - FOR ALL RECLAIMED TON *deposit* SUM OF RECLAIMED VALUE OF *deposit*
- VALUE OF RECLAIMED TOKEN DEPOSITS - NUMBER OF RECLAIMED TOKEN *deposit* MULTIPLIED BY 0.2 ton
- UNLOCK DEPOSITS WITH *votes* - the operation with the following results:
    - NEW `_requestedVotes` IS NOT NEGATIVE
    - *votes* IS NEW `_requestedVotes` PLUS TOTAL RECLAIMED VOTES
    - IF NEW `_requestedVotes` IS POSITIVE THAN NO NEW *deposit* IS RECLAIMABLE *deposit*
    - `_totalVotes` IS DECREASED BY THE DIFFERENCE BETWEEN *votes* and NEW `_requestedVotes`
    - FOR ALL *deposit* IN `deposits` amount OF *deposit* IS NOT NEGATIVE
    - FOR ALL *deposit* WHERE RECLAIMED VALUE OF *deposit* IS POSITIVE:
        - IF tokenID OF deposit IS ZERO THEN RECLAIMED VALUE OF *deposit* MUST be transferred to returnTo OF *deposit* using *transfer* method
        - IF tokenID OF *deposit* is NEITHER ZERO NOR ONE THEN *transfer* method OF address OF TIP IDED tokenID OF *deposit* MUST BE CALLED with the following arguments:
            - *value* - 0.2 ton
            - *dest* - returnTo OF *deposit*
            - *grams* - 0.1 ton

        - IF tokenID OF deposit IS ONE THEN **This case is omitted due to reasons described in the note above**
    - All the other elements are untouched
- REQUIRED VALUE FOR DEPOSIT UNLOCKING - VALUE OF RECLAIMED TON DEPOSITS PLUS VALUE OF RECLAIMED TOKEN DEPOSITS

Padawan Contract Functions: A Specification

Static member deployer

Requirements:

1. `deployer` MUST BE A VALID ADDRESS
2. `deployer` MUST NOT BE ZERO
3. `deployer` MUST IMPLEMENT `IDemiurge` interface
4. `deployer` NEVER CAN BE ALTERED

constructor()

Constructor can be invoked exclusively by the `deployer`. Sends `onPadawanDeploy` message to `deployer` with 1 ton payload.

Function Access Restrictions

1. *caller* MUST BE `deployer` OTHERWISE ERROR_INVALID_DEPLOYER

Outcome

1. `priceProvider` IS ZERO
2. `_wallet` IS ZERO
3. `tokenAccounts` IS EMPTY
4. `deposits` IS EMPTY
5. `_activeProposals` IS INITIAL ACTIVE PROPOSALS
6. `_requestedVotes` IS ZERO
7. `_totalVotes` IS ZERO
8. `_spentVotes` IS INITIAL SPENT VOTES
9. `_pendingDepositID` IS ZERO
10. `depools` IS INITIAL DEPOOLS

Messages

1. Messages
    a. `onPadawanDeploy`
        i. Recipient - `deployer`
        ii. Called - ever if no exceptions
        iii. Attributes
            1. value - 1 ton
        iv. Parameters
            1. Padawan public key

Errors

1. None but described above

initPadawan(address ownerAddress, address voteProvider, mapping(address => bool) depoolAddrs)

This function can be called exclusively by the `deployer` , sets the owner, price provider and depools.

Parameters

Parameter Name	Parameter Type	Description
`wallet`	`address`	Address of the `_wallet` to be used by the Padawan. MUST BE A VALID ADDRESS. MUST NOT BE ZERO.
`voteProvider`	`address`	Address of the _priceProvider to be used by the Padawan. MUST BE A VALID ADDRESS. MUST NOT BE ZERO. MUST IMPLEMENT `IPriceProvider` interface.
`depools`	depools	MAY BE EMPTY. EACH KEY IS A VALID ADDRESS.

Function Access Restrictions

1. *caller* MUST BE `deployer` OTHERWISE ERROR_UNAUTHORIZED_CALLER

Outcome

1. `_wallet` is wallet
2. `priceProvider` is voteProvider
3. `depools` is depools

Errors

None but described above

voteFor(address proposal, bool choice, uint32 votes)

This function is called by the wallet owner and paid by the caller. It allows the user to cast a vote (YES or NO) for a specified proposal, backing it with a specified number of votes.

Parameters

Parameter Name	Parameter Type	Description
`proposal`	`address`	Address of the Proposal smart contract the user intends to vote for. MUST BE A VALID ADDRESS. MUST BE NOT ZERO. MUST IMPLEMENT `IProposal` interface
`choice`	`boolean`	User's choice (YES or NO)
`votes`	`integer (uint32)`	Amount of votes the user casts to back his choice

Function Access Restrictions

1. *caller* MUST BE `_wallet` OTHERWISE ERROR_NOT_A_USER_WALLET.

Outcome

1. State is not changed

Errors

1. IF VOICES FOR proposal INCREASED BY votes IS GREATER THAN `_totalVotes` THAN ERROR_NOT_ENOUGH_VOTES

Messages

1. `voteFor` message is sent to:
    a. key - public key of the message
    b. choice - choice
    c. votes - votes
    d. No special value

confirmVote(uint64 pid, uint32 deposit)

This function is called by the Proposal smart contract to notify Padawan that the user's votes were accepted.

Parameters

Parameter Name	Parameter Type	Description
`pid`	`integer (uint64)`	Address of the Proposal smart contract the user has voted for
`deposit`	`integer (uint32)`	Amount of the user's votes that were accepted

Function Access Restrictions

1. *caller* MUST BE IN `_activeProposals` OTHERWISE EXCEPTION.

Outcome

1. VOICES FOR *caller* ARE INCREASED BY deposit
2. DECREASE SPENT VOTES FOR VOICES FOR *caller*
3. INCREASE SPENT VOTES FOR VOICES FOR *caller* INCREASED BY deposit
4. The change must be returned to *caller*

Errors

1. No but mentioned above

rejectVote(uint32 deposit, uint16 ec)

It's a callback method called by Padawan. Returns all the change and does not perform any state change (while it may alter some internal structures such as removal of zero values from hashmaps).

Parameters

Parameter Name	Parameter Type	Description
`pid`	`Integer (uint64)`	Unique proposal id
`deposit`	`integer (uint32)`	Amount of the user's votes that were rejected
`ec`	`Integer (uint16)`	Provide error code. May have one of the following values : ERROR_NOT_AUTHORIZED_VOTER, ERROR_VOTING_NOT_STARTED, ERROR_VOTING_HAS_ENDED, ERROR_VOTER_IS_NOT_ELIGIBLE

Function Access Restrictions

1. *caller* MUST BE IN `_activeProposals` OTHERWISE EXCEPTION.

Outcome

1. State not changed
2. Change must be returned to *caller*

Messages

1. *VoteRejected* event is emitted with the following parameters:
    a. *pid* - pid
    b. *votes* - deposit
    c. *ec* - ec

Й

Errors

1. No but mentioned above

depositTons(uint32 tons)

This function is called by an internal message and paid by the caller. It allows users to deposit TONs into Padawan. The TON deposit is converted into votes using the vote price obtained from the PriceProvider smart contract.

Parameter Name	Parameter Type	Description
`tons`	`integer (uint32)`	Amount of TONs to be deposited into Padawan (crystals, NOT nanocrystals)

Function access restrictions

1. *caller* MUST BE `_wallet` OTHERWISE ERROR_NOT_A_USER_WALLET.

Outcome

1. *new deposit* is created with the following attributes:
    a. *id* - MUST BE UNIQUE
    b. *tokenId* - MUST BE ZERO
    c. *returnTo* - MUST BE `_wallet`
    d. *amount* - MUST BE EQUAL TO tons
    e. *valuePerVote* - MUST BE ZERO
    f. *approved* - MUST BE TRUE
    g. *depool* - MUST BE ZERO
2. *new deposit* IS ADDED TO `deposits`
3. change must be returned to *caller*

Messages

1. *queryTonsPerVote* message is sent to `_priceProvider` with the following parameters:
    a. *value - DEPOSIT_TONS_FEE MINUS 0.1 ton*
    b. *queryId - id* OF *new deposit*

Errors

1. *id* OF *new deposit* MUST BE UNIQUE OTHERWISE EXCEPTION
2. *value* MUST BE GREATER OR EQUAL THAN tons INCREASED BY *DEPOSIT_TONS_FEE* OTHERWISE ERROR_MSG_VALUE_TOO_LOW.

depositTokens(address returnTo, uint256 tokenID, uint64 tokens)

This function is called by an internal message and paid by the caller. It allows to deposit and lock TIP3 tokens into a Padawan (Note: the tokens must be already transferred to Padawan's TIP3 token account before depositTokens is called). The function checks that the balance of Padawan's token wallet exceeds `tokens` argument. If so, the deposit is accepted and locked into the token account and converted to votes.

Parameter Name	Parameter Type	Description
`returnTo`	`address`	Address of the wallet where excess TIP3 tokens should be returned
`tokenID`	`integer (uint256)`	TokenId, corresponds to the standard address of the token root
`tokens`	`integer (uint64)`	Amount of the tokens to be deposited to the Padawan

Function access requirements

1. *caller* MUST BE `_wallet` OTHERWISE ERROR_NOT_A_USER_WALLET

Outcome

1. *new deposit* is created with the following attributes:
    a. *id* - MUST BE UNIQUE
    b. *tokenId* - MUST BE EQUAL TO tokenId
    c. *returnTo* - MUST BE returnTo
    d. *amount* - MUST BE EQUAL TO tokens
    e. *valuePerVote* - MUST BE ZERO
    f. *approved* - MUST BE FALSE
    g. *depool* - MUST BE ZERO
2. *new deposit* IS ADDED TO `deposits`
3. `_pendingDepositId` MUST BE *id* OF *new deposit*
4. change must be returned to *caller*

Messages

1. *getBalance_InternalOwner* message must be sent to *address* OF TIP IDED tokenId with the following arguments:

a. value MUST BE ZERO
b. *_answer_id* MUST BE ID OF *onGetBalance* function

Errors

1. *id* OF *new deposit* MUST BE UNIQUE OTHERWISE EXCEPTION
2. *value* MUST BE GREATER OR EQUAL THAN *DEPOSIT_TOKENS_FEE* OTHERWISE ERROR_MSG_VALUE_TOO_LOW.
3. `_pendingDepositId` MUST BE ZERO OTHERWISE ERROR_PENDING_DEPOSIT_ALREADY_EXISTS
4. STANDARD ADDRESS MADE FROM tokenId MUST BE IN `_tokenAccounts` OTHERWISE ERROR_ACCOUNT_DOES_NOT_EXIST

reclaimDeposit(uint32 votes)

This function is called by an internal message and paid by the caller. It allows users to return deposits (TONs, TIP3 tokens, DePool stakes) back to the user. However, LOCKED VOTES must be kept so LOCKED VOTES INCREASED BY votes MUST BE LESS OR EQUAL THAN `_totalVotes`. If the condition above is met the UNLOCK DEPOSIT must be performed.

Parameter Name	Parameter Type	Description
`votes`	`Integer (uint32)`	The amount of votes requested to be reclaimed.

Function access requirements

1. *caller* MUST BE `_wallet` OTHERWISE ERROR_NOT_A_USER_WALLET

Outcome

1. IF votes IS GREATER THAN `_totalVotes` MINUS LOCKED VOTES:
   a. `_requestedVotes` IS votes
2. IF votes IS LESS OR EQUAL THAN `_totalVotes` MINUS LOCKED VOTES:
   a. UNLOCK DEPOSITS votes

Messages

1. FOR ALL VOTED *proposal queryStatus* message MUST BE SENT with the following arguments:
   a. value - *QUERY_STATUS_FEE*

Errors

1. *value* MUST BE EQUAL OR GREATER THAN (NUMBER OF VOTED PROPOSALS * *QUERY_STATUS_FEE* + REQUIRED VALUE FOR DEPOSIT UNLOCKING) OTHERWISE ERROR_MSG_VALUE_TOO_LOW.
2. votes MUST BE LESS OR EQUAL THAN `_totalVotes` OTHERWISE ERROR_NOT_ENOUGH_VOTES.

updateStatus(uint64 pid, ProposalState state)

This function is called by the Proposal smart contract in response to Padawan's `queryStatus` request.

Parameter Name	Parameter Type	Description
`pid`	`Integer (uint64)`	Proposal ID. Not used.
`state`	`ProposalState`	The state of the proposal

Function Access Control

1. *caller* MUST BE IN `_activeProposals` OTHERWISE EXCEPTION.

Outcome

1. IF state IS AFTER Ended THEN
    a. DECREASE SPENT VOTES FOR VOICES FOR *caller* AND THEN:
        i. IF `_requestedVotes` IS LESS OR EQUAL THAN `_totalVotes` MINUS LOCKED VOTES:
            1. UNLOCK DEPOSITS `_requestedVotes`
2. OTHERWISE
    a. IF `_requestedVotes` IS LESS OR EQUAL THAN `_totalVotes` MINUS LOCKED VOTES:
        i. UNLOCK DEPOSITS `_requestedVotes`

Errors

1. No but mentioned above

createTokenAccount(address tokenRoot)

This function is called by the user. It allows the user to create a TIP3 token wallet controlled by Padawan. The created token wallet can be used to deposit TIP3 tokens, to be further converted into votes.

Parameter Name	Parameter Type	Description
`tokenRoot`	`address`	Address of the token smart contract that emits TIP3 tokens. MUST IMPLEMENT *ITokenRoot* interface

Function Access Control

1. *caller* MUST BE `_wallet` OTHERWISE ERROR_NOT_A_USER_WALLET

Outcome

1. *new tipAccount* MUST BE CREATED with the following parameters:
    a. *id* - MUST BE EQUAL TO tokenRoot
    b. *addr* - MUST BE ZERO
    c. *walletKey* - Padawan's own address CONVERTED TO VALUE
    d. *createdAt* - MUST BE UNIQUE (timestamp is an option)
    e. *balance* - MUST BE ZERO
2. New tipAccount MUST BE ADDED TO `tokenAccounts`

Messages

1. *deployEmptyWallet* message MUST BE SENT to tokenRoot with the following parameters:
    a. *value* MUST BE ZERO
    b. *answer_id* - id of Padawan's *onTokenWalletDeploy* function
    c. *workchain_id* - MUST BE ZERO
    d. *pubkey* - MUST BE ZERO
    e. *internal_owner* - Padawan's own address CONVERTED TO VALUE
    f. *grams* - 1 ton

Errors

1. *value* MUST BE GREATER OR EQUAL THAN *TOKEN_ACCOUNT_FEE* OTHERWISE *ERROR_MSG_VALUE_TOO_LOW*
2. IF *tokenRoot* WITH id OF *tokenRoot* EQUALS TO tokenRoot IN `tokenAccounts` THEN ERROR_TOKEN_ACCOUNT_ALREADY_EXISTS.

onTransfer(address source, uint128 amount)

This function is called by DePool to transfer the ownership of user stake to Padawan.

Parameter Name	Parameter Type	Description
`source`	`address`	Address of the DePool contract (?)
`amount`	`integer (uint128)`	Amount of stake to be transferred to Padawan (in nanoTONs)

Function access control

1. *caller* MUST BE IN `depools`

Outcome

1. *new deposit* is created with the following attributes:
    a. *id* - MUST BE UNIQUE
    b. *tokenId* - MUST BE EQUAL TO ONE
    c. *returnTo* - MUST BE source
    d. *amount* - MUST BE EQUAL TO amount
    e. *valuePerVote* - MUST BE ZERO
    f. *approved* - MUST BE FALSE
    g. *depool* - MUST BE *caller* ADDRESS CONVERTED TO VALUE
2. *new deposit* IS ADDED TO `deposits`
3. ACCEPT MUST BE EXECUTED AFTER REQUIRES

Messages

1. *queryTonsPerVote* message MUST BE SENT TO `_priceProvider` with the following parameters:
    a. *value* - 0.1 ton
    b. *queryId* - id OF *new deposit*

Errors

1. Nothing but mentioned above

receive()

Empty handler for fund receiving. Does nothing as the receiving is plain.

Function access control

1. No restrictions

Outcome

1. state IS NOT CHANGED

Errors

1. Nothing

fallback()

Empty handler for arbitrary messages. Does nothing.

Function access control

1. No restrictions

Outcome

1. state IS NOT CHANGED

Errors

1. Nothing

updateTonsPerVote(uint32 queryId, uint64 price)

Called as a callback from _priceProvider. Sets a new valuePerVote for a deposit identified by queryId. Also updates _totalVotes in regard to the updated price.

Parameter Name	Parameter Type	Description
`queryId`	`integer (uint32)`	id OF *deposit* to be priced
`price`	`integer (uint128)`	Deposit price (in nanotons)

Function access control

1. *caller* MUST BE `_priceProvider` OTHERWISE EXCEPTION

Outcome

1. WHEN *deposit* IN `deposits` WHERE id OF *deposit* IS queryId
    a. valuePerVote OF *deposit* MUST BE EQUAL TO price
    b. `_totalVotes` IS INCREASED BY ROUNDING DOWN OF DIVISION OF amount OF *deposit* BY price
    c. Change must be returned to caller

Errors

1. EXISTS *deposit* IN `deposits` WHERE id OF *deposit* IS queryId OTHERWISE *ERROR_DEPOSIT_NOT_FOUND*

updateTipsPerVote(uint32 queryId, uint64 price)

Called as a callback from _priceProvider. Sets a new valuePerVote for a deposit identified by queryId. Also updates _totalVotes in regard to the updated price.

Parameter Name	Parameter Type	Description
`queryId`	`integer (uint32)`	id OF *deposit* to be priced
`price`	`integer (uint128)`	Deposit price (in tokens)

Function access control

1. *caller* MUST BE `_priceProvider` OTHERWISE EXCEPTION

Outcome

1. WHEN *deposit* IN `deposits` WHERE id OF *deposit* IS queryId
    d. valuePerVote OF *deposit* MUST BE EQUAL TO price
    e. `_totalVotes` IS INCREASED BY ROUNDING DOWN OF DIVISION OF amount OF *deposit* BY price
    f. Change must be returned to caller

Errors

1. EXISTS *deposit* IN `deposits` WHERE id OF *deposit* IS queryId OTHERWISE *ERROR_DEPOSIT_NOT_FOUND*

onGetBalance(uint128 balance)

This function is a callback invoked by the token wallet to finish setting up the pending deposit. If the token wallet is not found in `_tokenAccounts` or there is no `_pendingDeposiIdt` it fails with an exception. As well the provided balance MUST BE GREATER OR EQUAL THAN the amount of pending deposit INCREASED BY the current balance of token wallet. Otherwise, `_pendingDepositId` is just destroyed. Afterwards, `_pendingDepositId` is always ZERO.

Parameter Name	Parameter Type	Description
`balance`	`integer (uint128)`	balance assigned to the pending deposit

Function access control

1. TIP ADDRESSED *caller* MUST EXIST OTHERWISE ERROR_ACCOUNT_DOES_NOT_EXIST

Outcome

1. `_pendingDepositId` MUST BE ZERO
2. *balance* OF TIP ADDRESSED *caller* MUST BE balance
3. IF balance IS GREATER OR EQUAL THAN *amount* OF PENDING DEPOSIT PLUS *balance* OF TIP ADDRESSED *caller* THEN:
    a. *approved* OF PENDING DEPOSIT IS TRUE
4. OTHERWISE:
    a. PENDING DEPOSIT MUST BE REMOVED FROM `deposits`

Messages

1. IF balance IS GREATER OR EQUAL THAN *amount* OF PENDING DEPOSIT PLUS *balance* OF TIP ADDRESSED *caller* THEN:
    a. *queryTipsPerVote* MESSAGE MUST BE SENT TO `_priceProvider` with the following parameters:
        i. *value* MUST BE ZERO
        ii. *queryId* IS `_pendingDepositId`
        iii. *tokenRoot* IS TIP IDED *caller*

Errors

1. PENDING DEPOSIT MUST EXIST OTHERWISE ERROR_DEPOSIT_NOT_FOUND

onTokenWalletDeploy(address wallet)

This is a callback function that must be invoked by the token root to finish setting up a token account bound to the provided wallet. The change must be returned.

Parameter Name	Parameter Type	Description
`wallet`	`address`	Address of the wallet to be bound

Function access control

1. TIP IDED *caller* MUST EXIST OTHERWISE ERROR_INVALID_ROOT_CALLER

Outcome

1. *walletKey* OF TIP IDED *caller* MUST BE EQUAL TO wallet
2. Change must be returned

Errors

1. Nothing but mentioned above

getDeposits()

Function access control

1. No restrictions

Outcome

1. No state change
2. Return - `deposits`

Errors

1. No errors

getTokenAccounts()

Function access control

1. No restrictions

Outcome

1. No state change
2. Return - `tokenAccounts`

Errors

1. No errors

getVoteInfo()

Function access control

1. No restrictions

Outcome

1. No state change
2. Return - the following tuple:
    a. *reqVotes* - `_requestedVotes`
    b. *totalVotes* - `_totalVotes`
    c. *lockedVotes* - LOCKED VOTES

Errors

1. No errors

getAddresses()

Function access control

1. No restrictions

Outcome

1. No state change
2. Return - the following tuple:
    a. *userWallet* - `_wallet`
    b. *priceProvider* - `_priceProvider`

Errors

1. No errors

getActiveProposals()

Function access control

1. No restrictions

Outcome

1. No state change
2. Return - `_activeProposals`

Errors

1. No errors

Proposal smart contract description

A Proposal is a smart contract that accumulates votes from Padawans and evaluates voting results. A Proposal is deployed by Demiurge by user request (deployProposal) and notifies Demiurge about its state.

State description of the Proposal contract

The full state of the Proposal contract may be described by the table below. It's important to mention that this state is using the external observer's point of view (some kind of God mode) so some elements may not be kept by the real implementation. For example, the real implementation may not keep the list of the token contracts but for the external observer such a list is still a part of the state.

Variable	Subvariable (if applicable)	Description
`_info`		Proposal info (refer to IBaseData.sol for details)

	`id`	Proposal ID MUST BE UNIQUE
	`start`	Time of the start of the voting MUST NOT BE ZERO
	`end`	Time of the end of the voting MUST NOT BE ZERO
	`options`	Proposal custom options - may be a combination of PROPOSAL_VOTE_SOFT_MAJORITY, PROPOSAL_VOTE_SUPER_MAJORITY, PROPOSAL_HAS_WHITELIST
	`totalVotes`	Maximum amount of votes that can be accepted by the proposal
	`description`	Proposal description
	`text`	Proposal text
	`voters`	List of addresses of smart contracts eligible to vote for this proposal
	`ts`	Proposal creation time as a timestamp
	`customData`	Proposal custom data as a TVMCell (to keep *.pdf* or other types of documents)
`_deployer`		Address of the deployer, supposed to be an instance of Demiurge
`_state`		Proposal status (ProposalState, how many votes were cast for and against the proposal)
	`state`	Internal state of the proposal
	`votesFor`	Amount of votes cast **for** the proposal

	votesAgainst	Amount of votes cast **against** the proposal
_haswhitelist		Defines whether or not the Proposal has a white list
_voters		White list of addresses of Padawans that are permitted to vote in the contest
_padawanSI		Padawan state image (used to further verify if a voting contract is indeed a Padawan)
_results		Voting results
	id	ID of the proposal
	passed	Indicates whether the proposal was passed or not
	votesFor	Amount of votes cast **for** the proposal
	votesAgaist	Amount of votes cast **against** the proposal
	totalVotes	Total amount of votes cast in the voting
	model	Vote counting model of the proposal
	ts	Time of the issuing of the results
_votecountmodel		Vote count model of the proposal (refer to Glossary.sol for details on VoteCountModel)

Special Stories

_state

This substate holds the information about the current state of the proposal and the number of votes cast for and against the proposal, respectively.

The following formal language is used for **_state:**

- PROPOSAL STATE for *state* - internal state of the proposal
- INITIAL PROPOSAL STATE IS EQUAL TO ZERO
- INITIAL VOTES FOR IS EQUAL TO ZERO
- INITIAL VOTES AGAINST IS EQUAL TO ZERO

Additional requirements:

- PROPOSAL STATE IS NOT NEGATIVE
- VOTES FOR IS NOT NEGATIVE
- VOTES AGAINST IS NOT NEGATIVE

_calculateVotes(uint32 yes, uint32 no, uint32 total)

This story is about a private function determining the outcome of the voting.

The following formal language is used in this story:

- PASSED is a boolean variable that determines whether the proposal has passed. The default value is FALSE.
    - IF `_voteCountModel` IS *Majority* AND votesFor IN `_state` IS GREATER THAN votesAgainst IN `_state` THEN PASSED IS TRUE
    - IF `_voteCountModel` IS *SoftMajority* AND votes FOR IN `_state MULTIPLIED BY` totalVotes IN `_info MULTIPLIED BY 10` IS GREATER OR EQUAL THAN totalvotes IN `_info` SQUARED PLUS votesAgainst IN `_state` MULTIPLIED BY (totalvotes IN `_state` + 20) THEN PASSED IS TRUE
    - IF `_voteCountModel` IS *SuperMajority* AND votesFOR IN `_state MULTIPLIED BY` totalVotes IN `_info MULTIPLIED BY 3` IS GREATER OR EQUAL THAN totalvotes IN `_info` SQUARED PLUS votesAgainst IN `_state` MULTIPLIED BY (totalvotes IN `_state` + 6) THEN PASSED IS TRUE
    - OTHERWISE PASSED IS FALSE

_tryEarlyComplete

This story describes a private method that is invoked to determine:

- Whether the voting on the proposal can be completed because a sufficient number of votes has been cast
- What is the outcome of the voting at the moment of its completion (passed or failed)

The following formal language is used for this story:

- COMPLETED is a boolean variable that shows whether the voting has been completed., The default value is FALSE
    - IF `_voteCountModel` IS *Majority* AND votesFor IN `_state MULTIPLIED BY 2` IS GREATER THAN _totalVotes IN `_info` THEN COMPLETED IS TRUE
    - IF `_voteCountModel` IS *Majority* AND votesAgainst IN `_state MULTIPLIED BY 2` IS GREATER THAN _totalVotes IN `_info` THEN COMPLETED IS TRUE
    - IF `_voteCountModel` IS *SoftMajority* AND votesFor IN `_state MULTIPLIED BY 2` IS GREATER THAN _totalVotes IN `_info` THEN COMPLETED IS TRUE
    - IF `_voteCountModel` IS *SoftMajority* AND votesAgainst IN `_state MULTIPLIED BY 2` IS GREATER THAN _totalVotes IN `_info` THEN COMPLETED IS TRUE
    - IF `_voteCountModel` IS *SoftMajority* AND votesAgainst IN `_state MULTIPLIED BY 2` IS GREATER THAN _totalVotes IN `_info` THEN COMPLETED IS TRUE
    - IF `_voteCountModel` IS *SuperMajority* AND votesFor IN `_state MULTIPLIED BY 3` IS GREATER THAN _totalVotes IN `_info` MULTIPLIED BY 2 THEN COMPLETED IS TRUE
    - IF `_voteCountModel` IS *SuperMajority* AND votesAgainst IN `_state MULTIPLIED BY 3` IS GREATER THAN _totalVotes THEN COMPLETED IS TRUE
    - OTHERWISE COMPLETED IS FALSE
- PASSED is a boolean variable determining whether the proposal voted on was passed. The default value is FALSE.
    - IF `_voteCountModel` IS *Majority* AND votesFor IN `_state MULTIPLIED BY 2` IS GREATER THAN _totalVotes IN `_info` THEN PASSED IS TRUE
    - IF `_voteCountModel` IS *SoftMajority* AND votesFor IN `_state MULTIPLIED BY 2` IS GREATER THAN _totalVotes IN `_info` THEN PASSED IS TRUE
    - IF `_voteCountModel` IS *SuperMajority* AND votesFor IN `_state MULTIPLIED BY 3` IS GREATER THAN _totalVotes IN `_info` MULTIPLIED BY 2 THEN PASSED IS TRUE
    - OTHERWISE PASSED IS FALSE

_transit (ProposalState state)

This is a private function (method?) that implements IInfoCenter interface.

FOR state of `_state`, an `onStateUpdate` MESSAGE MUST BE SENT TO Demiurge THAT MUST IMPLEMENT WITH VALUE {value: 0.2 ton, bounce:true} (`state`).

_wrapUp

This story describes an internal function called to determine whether the proposal can be completed and, if completed, retrieve the outcome of the voting (passed or failed).

The formalized language here:

- CURRENT TIME - the timestamp of the moment when **_wrapUp** is called
- PROPOSAL COMPLETED is a boolean variable that determines whether the Proposal is completed. The default value is FALSE.
- PROPOSAL PASSED is a boolean variable that determines whether the Proposal has passed. The default value is FALSE.

_wrapUp, when invoked, operates as follows:

A. It determines the values for PROPOSAL COMPLETED and PROPOSAL PASSED, whereby:

1. IF CURRENT TIME IS GREATER THAN end IN `_info` THEN PROPOSAL COMPLETED IS TRUE AND _calculateVotes is called:
    a. IF PASSED returned by _calculateVotes IS TRUE THEN PROPOSAL PASSED IS TRUE
    b. OTHERWISE PROPOSAL PASSED IS FALSE
2. IF CURRENT TIME IS LESS OR EQUAL THAN end IN `_info` THEN _tryEarlyComplete is called:
    a. IF COMPLETED RETURNED BY _tryEarlyCompleted IS TRUE THEN PROPOSAL COMPLETED IS TRUE OTHERWISE PROPOSAL COMPLETED IS FALSE
    b. IF PASSED returned by _tryEarlyCompleted IS TRUE THEN PROPOSAL PASSED IS TRUE OTHERWISE PROPOSAL COMPLETED IS FALSE

B. IF PROPOSAL COMPLETED IS TRUE THEN `_transit` IS CALLED WITH *ProposalState.Ended* THEN **finalize** IS called for PROPOSAL PASSED WITH DEF_COMPUTE_VALUE

buildStateInit

This story is about the conversion of an image represented by TVMCell into another TVMCell that can be used for contact initialization.

Formalized language here:

- STATE FROM *image* FOR *name* WITH *key* - the result of subsequent call of the following operations:
    - *image* converted to slice
    - first reference[11] is taken from the resulting slice
    - *tvm.buildStateInit* is called with the following arguments:

[11] Each slice has up to four references. The first one is taken. If the image is broken and does not have any references the exception is raised

        - *contr* - name
        - *varInit* - the following list of variables:
            - *deployer* - address of the Demiurge contract itself
        - *pubkey* - key
        - *code* - the reference received at the point above
    - The resulting *TVMCell* is the outcome of the operation

Proposal Contract Functions: A Specification

Static member deployer

Requirements

1. `deployer` MUST BE A VALID ADDRESS
2. `deployer` MUST NOT BE ZERO
3. `deployer` MUST IMPLEMENT `IDemiurge` interface
4. `deployer` NEVER CAN BE ALTERED

constructor

constructor may be only called by `deployer`.

Additional requirement: the deployer must implement IInfoCenter interface.

Parameters

The constructor has no parameters.

Function Access Requirements

1. *caller* MUST BE `deployer` OTHERWISE EXCEPTION

Outcome

1. _state IS MODIFIED WHEREBY
    a. PROPOSAL STATE IS set to New
    b. VOTES FOR IS INITIAL VOTES FOR
    c. VOTES AGAINST IS INITIAL VOTES AGAINST
2. _deployer is deployer
3. _haswhitelist IS EMPTY
4. _voters IS EMPTY

5. _padawanSI IS EMPTY
6. _results IS EMPTY
7. _voteCountModel IS EMPTY

Messages:

1. onProposalDeploy state from
    a. Recipient - `deployer`
    b. Attributes
        i. Value: DEF_RESPONSE_VALUE (0.03 ton)
    c. Parameters - none.

initProposal(ProposalInfo pi, TvmCell padawanSI)

This function can be invoked by the Demiurge smart contract that deployed the proposal. It sets proposal information, vote count model, and the white list (if the proposal is designated to have one).

Parameters

Parameter Name	Parameter Type	Description
`pi`	`ProposalInfo`	ID of the proposal MUST NOT BE ZERO
`padawanSI`	`TvmCell`	Padawan state image

Function access requirements

1. *caller* MUST BE `deployer` OTHERWISE EXCEPTION

Outcome

1. `_info` IS pi
2. `_padawanSI` IS padawanSI
3. _voteCountModel
    a. IF IN `options` OF pi PROPOSAL_VOTE_SOFT_MAJORITY IS TRUE THEN `_voteCountModel` IS VoteCountModel.SoftMajority
    b. IF IN `options` OF pi PROPOSAL_VOTE_SUPER_MAJORITY IS TRUE THEN `_voteCountModel` IS VoteCountModel.SuperMajority
    c. OTHERWISE `_voteCountModel` IS VoteCountModel.Majority

4. IF IN `options` OF pi PROPOSAL_HAS_WHITELIST IS TRUE THEN `_hasWhiteList` IS TRUE
    a. If _hasWhiteList IS TRUE THEN `_voters` IS voters OF pi.
5. `state` OF _state MUST BE ProposalState.OnVoting

wrapUp()

This function can be called by any smart contract via an internal message. It asks the Proposal to update its status.

Parameters: none

Outcome

1. **_wrapUp** internal function is called (see Special Stories for details on **_wrapUp**)

Message

1. A `transfer` message is sent to the caller RETURNING THE CHANGE:
    a. value: 0 ton
    b. bounce: false
    c. flag: 64

It invokes a private _wrapUp function which returns two boolean values: **completed** and **passed**.

voteFor(uint256 key, bool choice, uint32 deposit)

This function implements the SMV algorithm to receive Yes or No votes. It can be called by Padawans.

Parameters

Parameter Name	Parameter Type	Description
`key`	`integer (uint256)`	Padawan public key (used to check whether the calling contract is indeed a Padawan) MUST_NOT_BE_ZERO

`choice`	`boolean`	The Padawan's vote (yes or no) MUST_NOT_BE_ZERO
`deposit`	`integer (uint32)`	The number of votes cast by the Padawan MUST_NOT_BE_ZERO MUST_NOT_BE_NEGATIVE

Function access restrictions

This function can only be called by Padawans.

In addition, the caller must implement the IPadawan interface.

1. *caller* MUST BE padawanAddress OTHERWISE ERROR_NOT_AUTHORIZED_VOTER
2. *caller* MUST IMPLEMENT IPadawan interface

Outcome

1. IF *ec* IS MORE THAN THAN ZERO THEN `rejectVote` MESSAGE MUST BE SENT TO *caller* with the following parameters (CHANGE IS RETURNED):
    a. id IN `_info` - proposal ID
    b. *deposit* - deposit value
    c. *ec* - exit code
2. OTHERWISE
    a. `confirmVote` MESSAGE MUST BE SENT TO *caller* with the following parameters (CHANGE IS RETURNED):
        i. id IN `_info` - proposal ID
        ii. *deposit* - deposit value
    b. IF *choice* IS TRUE THEN votesFor IN `_state` IS INCREASED BY *votes*.
    c. OTHERWISE votesAgainst IN `_state` IS INCREASED BY *votes*.
3. **_wrapUp** is called to update proposal state.

Errors

1. IF ADDRESS OF STATE FROM `_padawanSI` FOR *Padawan* WITH key IS NOT *caller* THEN ec IS ERROR_NOT_AUTHORIZED_VOTER
2. IF CURRENT TIME IS LESS THAN start IN `_info` THEN ec IS ERROR_VOTING_NOT_STARTED
3. IF CURRENT TIME IS MORE THAN end IN `_info` THEN ec IS ERROR_VOTING_NOT_STARTED
4. IF _hasWhiteList IS TRUE AND *caller* IS NOT IN _voters THEN ec IS ERROR_VOTER_IS_NOT_ELIGIBLE

finalize(bool passed)

The function is called by *voteFor* and *wrapUp*. It retrieves the voting results and publishes them.

Parameter Name	Parameter Type	Description
`passed`	`boolean`	PROPOSAL PASSED value

Function Access Restrictions

1. *caller* MUST BE *this*

Outcome

1. `_results` are retrieved as VotingResults with the following components:
    a. id IN `_info` - proposal ID
    b. PROPOSAL PASSED - voting outcome
    c. votesFor IN `_state` - the amount of votes cast for the proposal
    d. votesAgainst IN `_state` - the amount of votes cast against the proposal
    e. totalVotes IN `_info` - the total amount of votes that can be cast for the proposal
    f. _voteCountModel - the vote count model of the proposal
    g. uint32(now) - timestamp of the results
2. *state* is PROPOSAL PASSED
3. **_transit(*state*)** is called
4. **ProposalFinalized** event is called with `_results`

Messages

1. A *reportResults* message IS SENT TO `_deployer` with:
    a. value: 1 ton
    b. results MUST BE `_results`

queryStatus()

This function is called by a Padawan to retrieve the proposal state.

Parameters

This function has no parameters.

Function Access Control

This function can only be called by Padawans. In addition, IPadawan interface must be implemented.

Outcome

1. *updateStatus* message is sent with the following content:
    a. *id* - proposal ID
    b. state - proposal state

getID()

A public function that returns the id of the proposal.

Function access restrictions

1. No restrictions

Outcome

1. State is not changed
2. Return value is *id* OF `_info`

getVotingResults()

A public function that returns `_results`.

Function Access Requirements

1. *state* of `_state` MUST BE MORE THAN Ended

Outcome

1. State is not changed
2. The return value is `_results`

getInfo()

A public function that returns `_info`.

Function Access Requirements

1. No restrictions

Outcome

1. State is not changed
2. The return value is `_info`

getCurrentVotes()

A public function that returns the current values of `votesFor` and `votesAgainst` as uint32.

Function Access Requirements

1. No restrictions

Outcome

1. State is not changed
2. Return value is the pair of:
    a. *votesFor* OF `_state`
    b. *votesAgainst* OF `_state`

getProposalData()

A public function that returns the proposal's `_info` and `_state`.

Function Access Requirements

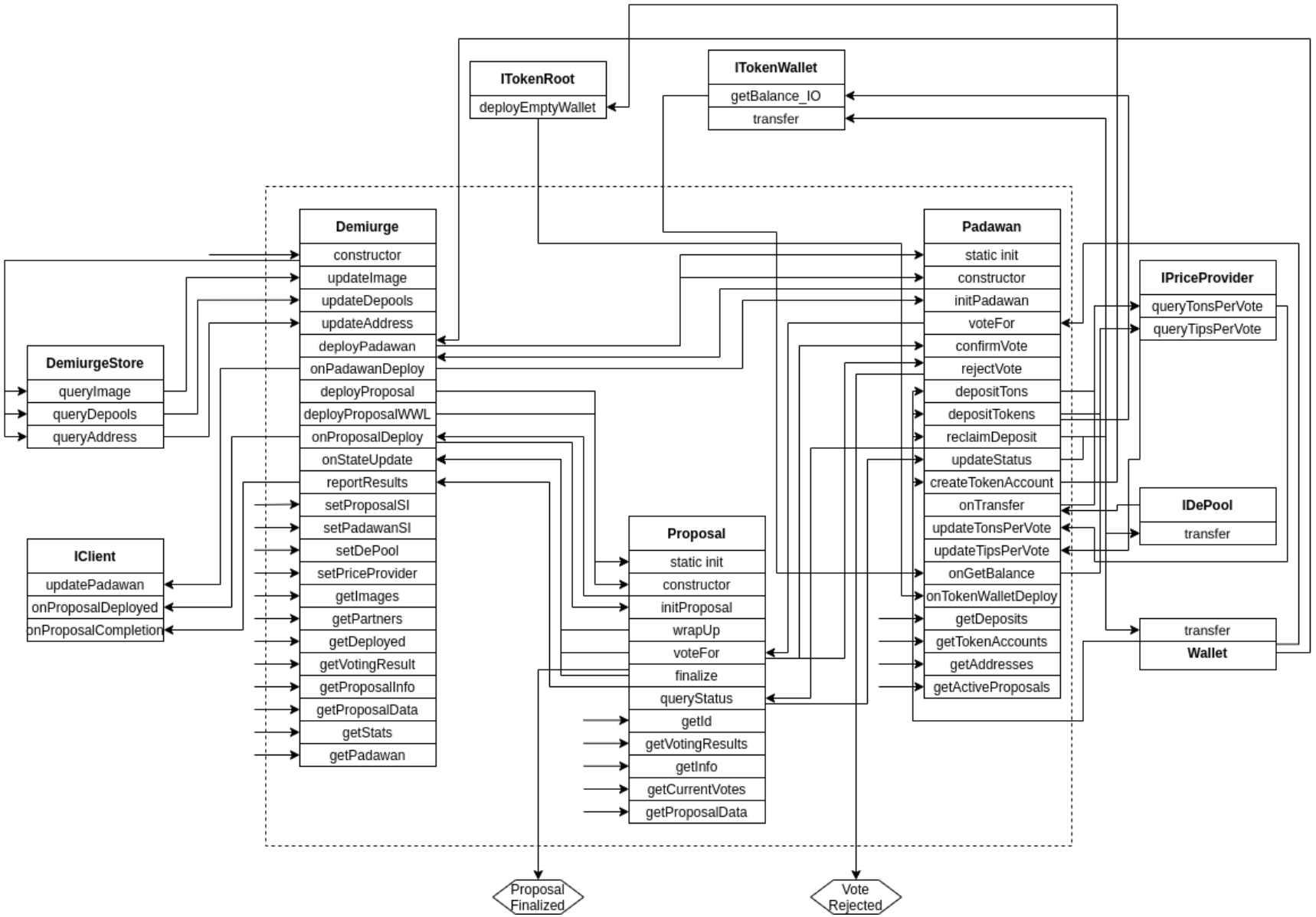
1. No restrictions

Outcome

1. State is not changed
2. Return value is a pair of:
    a. `_info`
    b. `_state`

Cross-functional specification

1. All the cross-function calls are executed either as internal messages explicitly sent by origin functions or as internal messages sent by the system
2. Any calls not mentioned in the present specification MUST not exist
3. Recursive calls are not allowed
4. Number of calls made by each function MUST be fixed (no calls in loops)
5. The diagram below illustrates all the internal and external messages in the system


ITokenRoot
deployEmptyWallet
ITokenWallet
getBalance_IO
transfer
Demiurge
constructor
updateImage
updateDepools
updateAddress
deployPadawan
onPadawanDeploy
deployProposal
deployProposalWWL
onProposalDeploy
onStateUpdate
reportResults
setProposalSI
setPadawanSI
setDePool
setPriceProvider
getImages
getPartners
getDeployed
getVotingResult
getProposalInfo
getProposalData
getStats
getPadawan
Padawan
static init
constructor
initPadawan
voteFor
confirmVote
rejectVote
depositTons
depositTokens
reclaimDeposit
updateStatus
createTokenAccount
onTransfer
updateTonsPerVote
updateTipsPerVote
onGetBalance
onTokenWalletDeploy
getDeposits
getTokenAccounts
getAddresses
getActiveProposals
IPriceProvider
queryTonsPerVote
queryTipsPerVote
IDePool
transfer
transfer
Wallet
DemiurgeStore
queryImage
queryDepools
queryAddress
IClient
updatePadawan
onProposalDeployed
onProposalCompletion
Proposal
static init
constructor
initProposal
wrapUp
voteFor
finalize
queryStatus
getId
getVotingResults
getInfo
getCurrentVotes
getProposalData
Proposal Finalized
Vote Rejected

Specification for special cases

Gas specification

1. No special requirements for acceptance are set unless it is explicitly stated otherwise
2. Each function must spend less than *10 000* of gas

Code-upgrade specification

1. Demiurge contract must support *upgrade* message that:
    a. Has the single parameter state of *TVMCell* type
    b. *Caller* must be Demiurge creator
    c. The following sequence of actions must be performed:
        i. The first reference of state must be taken and called as newcode
        ii. Gas must be accepted
        iii. All the changes must be committed
        iv. *tvm.setcode()* must be called with newcode as a parameter
        v. *tvm.setCurrentcode()* must be called with newcode as a parameter
    d. *onCodeUpgrade* message must be sent to itself
2. Demiurge contract must support *onCodeUpgrade* message that:
    a. Resets all the storage
    b. Queries *DemiurgeStore* for:
        i. Padawan and Proposal images
        ii. DePools
        iii. Price provider

Correctness

Plain correctness

The statements described below MUST BE TRUE before and after handling of each message. Inside the message handling some of these statements may temporarily become false. Each of the statements can be checked instantly, without any check for the future.

1. `_padawanSI` IS NOT ZERO EQUALS TO `CHECK_PADAWAN` IS TRUE
2. `_proposalSI` IS NOT ZERO EQUALS TO `CHECK_PROPOSAL` IS TRUE
3. `_priceProvider` IS NOT ZERO EQUALS TO `CHECK_PRICE_PROVIDER` IS TRUE
4. IF `_deployedPadawans` IS NOT EMPTY THEN CHECK_PASSED
5. FOR ALL *padawan1*, *padawan2* IN `_deployedPadawans`:
    a. *padawanId* OF *padawan1* MUST NOT BE EQUAL TO *padawanId* OF *padawan2*
    b. *addr* OF *padawan1* MUST NOT BE EQUAL TO *addr* OF *padawan2*

6. FOR ALL *padawan* IN `_deployedPadawans`:
    a. *userWalletAddress* OF *padawan* MUST NOT BE ZERO
    b. *addr* OF *padawan* MUST NOT BE ZERO'
    c. CONTRACT OF *addr* OF padawan MUST IMPLEMENT *IPadawan* INTERFACE
7. IF `_deployedProposals` IS NOT EMPTY THEN CHECK_PASSED
8. FOR ALL *proposal1*, *proposal2* IN `_deployedProposals`:
    a. *address* OF *proposal1* MUST NOT BE EQUAL TO *address* OF *proposal2*
    b. *id* OF *proposal1* MUST NOT BE EQUAL TO *id* OF *proposal2*
    c. *ts* OF *proposalData* OF *proposal1* MUST NOT BE EQUAL TO *ts* OF *proposalData* OF *proposal2*
9. FOR ALL *proposal* IN `_deployedProposals` (OF *proposal* is omitted in this section):
    a. *address* MUST NOT BE ZERO
    b. CONTRACT OF *address* MUST IMPLEMENT *IProposal* INTERFACE
    c. *id* OF *proposalInfo* MUST BE EQUAL TO *id*
    d. *start* OF *proposalInfo* MUST BE LESS THAN *end* OF *proposalInfo*
    e. *start* OF *proposalInfo* MUST BE POSITIVE
    f. *options* OF *proposalInfo* CANNOT HAVE BOTH *PROPOSAL_VOTE_SOFT_MAJORITY* AND *PROPOSAL_VOTE_SUPER_MAJORITY* BITS
    g. *totalVotes* OF *proposalInfo* MUST BE POSITIVE
    h. FOR ALL *voter* IN *voters* OF *proposalInfo* :
        i. *voter* MUST NOT BE ZERO
    i. *ts* OF *proposalInfo* MUST BE POSITIVE
    j. *id* OF proposalData MUST BE EQUAL TO *id*
    k. *ProposalState* OF *proposalData* MUST BE EITHER:
        i. *New*
        ii. *OnVoting*
        iii. *Ended*
        iv. *Passed*
        v. *Failed*
    l. *userWalletAddress* OF *proposalData* MUST NOT BE ZERO
    m. *addr* OF *proposalData* MUST BE EQUAL TO *address*
    n. *ts* OF proposalData MUST BE POSITIVE
10. IF `depools` IS NOT EMPTY THEN `CHECK_DEPOOLS` IS TRUE
11. FOR ALL *depool1*, *depool2* IN `depools` :
    a. *address* OF *depool1* MUST NOT BE EQUAL TO *address* OF *depool2*
12. FOR ALL *depool* in `depools`:
    a. CONTRACT OF *address* OF *depool* MUST IMPLEMENT *IDePool* INTERFACE
13. IF `demiStore` IS NOT ZERO THAN `demiStore` MUST BE A *DemiurgeStore* CONTRACT
14. FOR ALL *votingResult1*, *votingResult2* IN `_votingResults`:
    a. *id* OF *votingResult*1 MUST NOT BE EQUAL TO *id* OF *votingResult2*
15. FOR ALL *votingResult* IN `_votingResults`:

a. EXISTS *proposal* IN `_deployedProposals` WHERE *id* OF *proposal* IS EQUAL TO *id* OF *votingResult* AND:
    i. *totalVotes* OF *votingResult* MUST BE EQUAL TO *totalVotes* OF *proposal*
    ii. *ts* OF *votingResult* MUST BE GREATER THAN *ts* OF *proposal*
    iii. IF *model* OF *votingResult* IS *VoteCountModel.SoftMajority* THEN LOGICAL OR OF *options* OF *proposalInfo* OF *proposal* AND *PROPOSAL_VOTE_SOFT_MAJORITY* IS NOT ZERO
    iv. IF *model* OF *votingResult* IS *VoteCountModel.SuperMajority* THEN LOGICAL OR OF *options* OF *proposalInfo* OF *proposal* AND *PROPOSAL_VOTE_SUPER_MAJORITY* IS NOT ZERO
    v. *ts* OF *votingResult* MUST BE GREATER THAN *ts* OF *proposal*
b. *votesFor* OF *votingResult* PLUS *votesAgainst* OF *votingResult* IS LESS OR EQUAL TO *totalVotes* OF *votingResult*
c. IF *passed* OF *votingResult* IS TRUE THEN *model* OF *votingResult* IS EITHER *VoteCountModel.Majority* OR *VoteCountModel.SoftMajority* OR *VoteCountModel.SuperMajority*
d. IF *passed* OF *votingResult* IS TRUE AND *model* OF *votingResult* IS *VoteCountModel.Majority* THEN *votesFor* OF *votingResult* IS GREATER THAN *votingAgainst* OF *votingResult*
e. IF *passed* OF *votingResult* IS TRUE AND *model* OF *votingResult* IS *VoteCountModel.Majority* THEN SOFT MAJORITY OF *votesFor* OF *votingResult*, *votesAgainst* OF *votingResult*, *totalVotes* OF *votingResult* IS REACHED
f. IF *passed* OF *votingResult* IS TRUE AND *model* OF *votingResult* IS *VoteCountModel.SuperMajority* THEN SUPER MAJORITY OF *votesFor* OF *votingResult*, *votesAgainst* OF *votingResult*, *totalVotes* OF *votingResult* IS REACHED

16. FOR ALL *padawan* IN `_deployedPadawans` AND *Padawan* WHERE CONTRACT OF *padawan* IS *Padawan* (OF *Padawan* is omitted in the present section):
    a. `deployer` MUST BE EQUAL TO THE ADDRESS OF *Demiurge*
    b. IF `_wallet` IS NOT ZERO THEN `_wallet` IS EQUAL TO *userWalletAddress* OF *padawan*
    c. IF `_priceProvider` IS NOT ZERO THEN `_priceProvider` IS EQUAL TO `_priceProvider` OF *Demiurge*
    d. FOR ALL *tokenAccount1*, *tokenAccount2* IN `_tokenAccounts`:
        i. *tokenRoot* OF *tokenAccount1* MUST NOT BE EQUAL TO *tokenRoot* OF *tokenAccount2*
        ii. *address* OF *tokenAccount1* MUST NOT BE EQUAL TO *address* OF *tokenAccount2*
    e. FOR ALL *tokenAccount* IN `_tokenAccounts`:
        i. *tokenRoot* OF *tokenAccount* MUST NOT BE ZERO
        ii. *tokenRoot* OF *tokenAccount* MUST IMPLEMENT *ITokenRoot* INTERFACE

iii. *walletKey* OF *tokenAccount* MUST NOT BE ZERO
iv. *createdAt* OF *tokenAccount* MUST BE POSITIVE

f. FOR ALL *deposit1*, *deposit2* IN `deposits` :
   i. *id* OF *deposit1* MUST NOT BE EQUAL TO *id* OF *deposit2*
   ii. IF *approved* OF *deposit1* IS FALSE THEN *approved* OF *deposit2* IS TRUE

g. FOR ALL *deposit* IN `deposits`:
   i. *returnTo* OF *deposit* MUST NOT BE ZERO
   ii. *amount* OF *deposit* MUST NOT BE NEGATIVE
   iii. *valuePerVote* OF *deposit* MUST NOT BE NEGATIVE
   iv. *depool* OF *deposit* IS NOT ZERO EQUALS TO *tokenId* OF *deposit* IS ONE
   v. IF *depool* OF *deposit* IS NOT ZERO THEN:
      1. EXISTS *_depool* IN `depools` WHERE *address* OF *_depool* IS *depool* AND *enabled* OF *_depool* IS TRUE

h. FOR ALL *proposal1*, *proposal2* IN `_activeProposals` :
   i. *address* OF *proposal1* MUST BE NOT EQUAL TO *address* OF *proposal2*

i. FOR ALL *proposal* IN `_activeProposals` :
   i. EXISTS *deployedProposal* IN `_deployedProposals` WHERE *id* OF *proposal* EQUALS TO *id* OF *deployedProposal*
   ii. *votes* FOR *proposal* IS NOT NEGATIVE

j. `_requestedVotes` IS NOT NEGATIVE

k. `_totalVotes` IS NOT NEGATIVE

l. `_totalVotes` IS LESS OR EQUAL THAN SUM OF ALL `deposits` ACCUMULATED BY *amount* DIVIDED BY *pricePerVote*

m. FOR ALL *spentVote* IN `_spentVotes` :
   i. *vote* in *spentVote* IS NOT NEGATIVE
   ii. *vote* in *spentVote* IS LESS OR EQUAL TO `_totalVotes`
   iii. *quantity* in *spentVote* IS NOT NEGATIVE

n. IF `_pendingDepositID` IS EQUALS TO ZERO THEN:
   i. FOR ALL *deposit* IN `deposits`:
      1. *approved* OF *deposit* IS TRUE

o. IF `_pendingDepositID` IS NOT ZERO THEN:
   i. EXISTS *deposit* IN `deposits`:
      1. *approved* OF *deposit* IS FALSE
      2. `_pendingDepositId` EQUALS TO *id* OF *deposit*

17. FOR ALL *proposal* IN `_deployedProposals` AND *Proposal* WHERE CONTRACT OF *proposal* IS *Proposal* (OF *Proposal* is omitted in the present section):
    a. `deployer` MUST BE EQUAL TO THE ADDRESS OF *Demiurge*
    b. *state* OF `_state` MUST BE EITHER:
       i. New
       ii. OnVoting

iii. *Ended*
iv. *Passed*
v. *Failed*

c. *votesFor* OF `_state` MUST BE NOT NEGATIVE
d. *votesAgainst* OF `_state` MUST BE NOT NEGATIVE
e. IF *state* OF `_state` IS EITHER *New*, *OnVoting*, *Ended* THEN `_results` IS ZERO
f. IF *state* OF `_state` IS EITHER *Passed*, Failed THEN `_results` IS NOT ZERO
g. IF `_results` IS ZERO THEN:
    i. NO *votingResult* IN `_votingResults` EXISTS WHERE:
        1. *id* OF *votingResults* EQUALS TO *id* OF `_info`
h. IF `_results` IS NOT ZERO THEN:
    i. EXISTS *votingResult* IN `_votingResults` EXISTS WHERE:
        1. *votingResults* EQUALS TO `_results`

Eventual correctness

Eventual correctness is a set of statements that can be wrong at the moment (after completion of certain functions) but become true either after the invoking of the certain function (that ought to be invoked at some time) or just happens eventually. To describe such a behaviour the following statements are introduced:

- AFTER <method> WILL <statement> - anytime after the invocation of the <method> (related to one of the contracts being specified) the <statement> must be considered as TRUE. It's required that <method> must be called eventually
- EVENTUALLY <statement> - the method can be wrong for a moment but eventually it becomes TRUE.

The following eventual statements are TRUE for the correct state of the set of smart contracts being discussed:

1. EVENTUALLY :
    a. FOR ALL *proposal* IN `_deployedProposals` AND *Proposal* WHERE CONTRACT OF *proposal* IS *Proposal* (OF *Proposal* is omitted in the present section):
        i. EVENTUALLY EXISTS *votingResult* in `_votingResults` WHERE:
            1. *id* OF *_info* OF *proposal* EQUALS TO *id* OF *votingResults*
            2. *votingResult* EQUALS TO *_results* OF *proposal*
2. AFTER *initPadawan* WILL:
    a. `_wallet` IS NOT ZERO
    b. EXISTS *padawan* IN `_depolyedPadawans` WHERE:
        i. CONTRACT OF *padawan* IS *Padawan*
        ii. *userWalletAddress* OF *padawan* EQUALS TO `_wallet`

iii. `_priceProvider` IS `_priceProvider` OF *Demiurge*
iv. `depools` IS `depools` OF Demiurge

3. AFTER *initProposal* WILL:
    a. EXISTS *proposal* IN `_depolyedProposals` WHERE:
        i. CONTRACT OF *proposal* IS *Proposal*
        ii. `_info` IS *proposalInfo* OF *proposal*
        iii. `_padawanSI` IS `_padawanSI` OF *Demiurge*
        iv. IF LOGICAL OR OF options OF `_info` AND *PROPOSAL_VOTE_SOFT_MAJORITY* THEN `_votecountmodel` IS *VoteCountModel.SoftMajority*
        v. IF LOGICAL OR OF options OF `_info` AND *PROPOSAL_VOTE_SUPER_MAJORITY* THEN `_votecountmodel` IS *VoteCountModel.SuperMajority*
        vi. OTHERWISE `_votecountmodel` IS *VoteCountModel.Majority*
        vii. `_hasWhiteList` IS LOGICAL OR OF `_info` AND PROPOSAL_HAS_WHITELIST

State machine (projections)

After a set of experiments, a finite state machine was selected as a basic tool for scenario building. However, the full set of states is too huge for any kind of handling so the projections of states to a rather small set of hypersurfaces is used for forming the scenarios. It's important to ensure that the selected set of hypersurfaces is full and covers all the dimensions of the original state. Thus the overall number of scenarios is a sum of scenarios for each hypersurface rather than multiplication as it would be for the complete state.

For each hypersurface the conventional way of finite state machine representation is used where squares illustrate the different state, arrows - possible transitions between states and titles for these arrows - conditions for transitions. It is worth noting that conditions have heterogeneous form and consist of an external event (in most cases) as well as a logical condition (also, in most cases).

This particular contract has degenerate projections only - each of them consists on only one state with some attributes so all the moves will be to itself (with proper attribute changing).

Additionally some states have attributes (such as "balance") and its evolution during transitions may be represented as the second title for arrows.

Below all the hypersurfaces are listed as well as their state-condition diagrams:

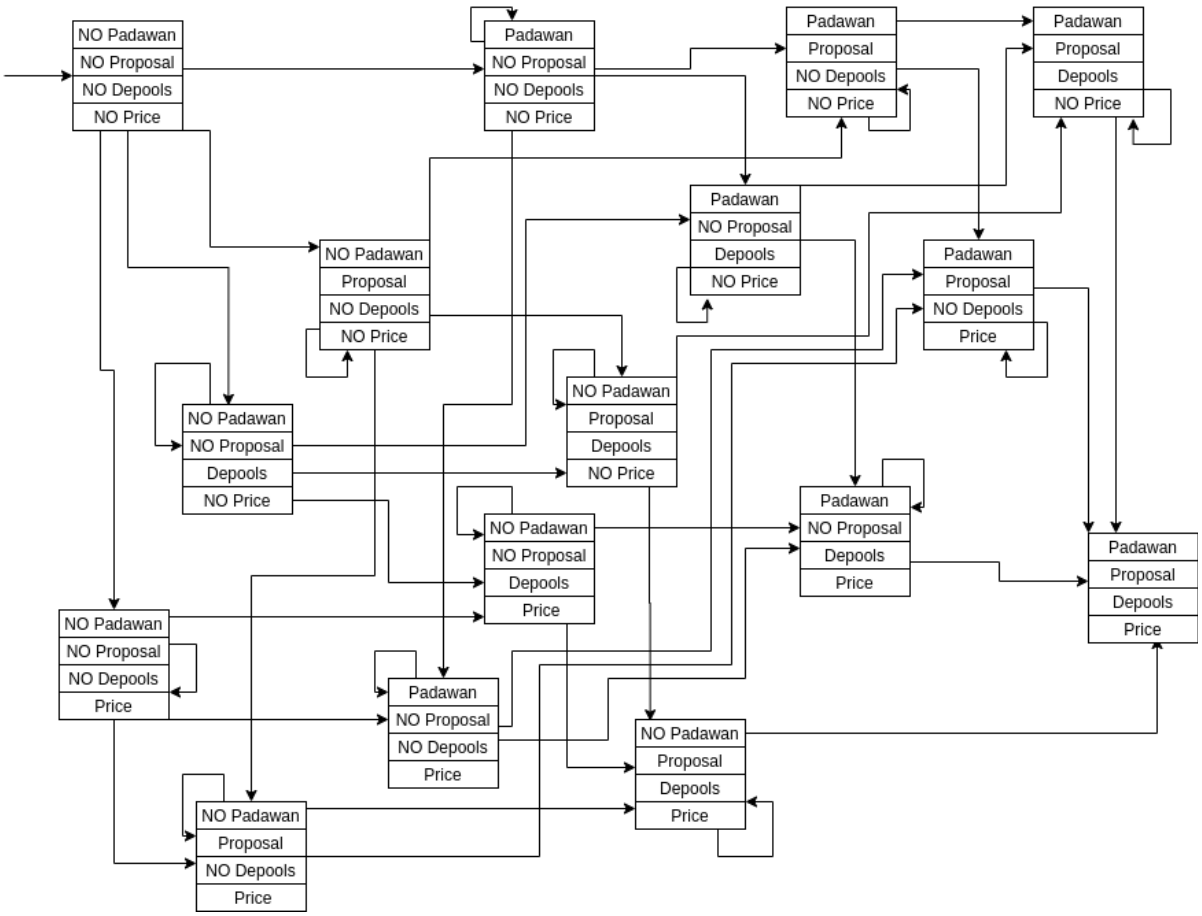
DM_images:

This projection represents external elements of the *Demiurge* such as images of *Padawan* and *Proposal*, *DePools* and *PriceProvider*.

Covers the following variables of the Demiurge contract:

- `_padawanSI`
- `_proposalSI`
- `_priceProvider`
- `depools`
- `CHECK_PADAWAN`
- `CHECK_PROPOSAL`
- `CHECK_DEPOOLS`
- `CHECK_PRICE_PROVIDER`

Please note that as the visual representation of this projection is rather large the attributes mentioned above are omitted as well as the movement methods that are, in all the cases but constructor, a pair of corresponding *update** and *set** methods.

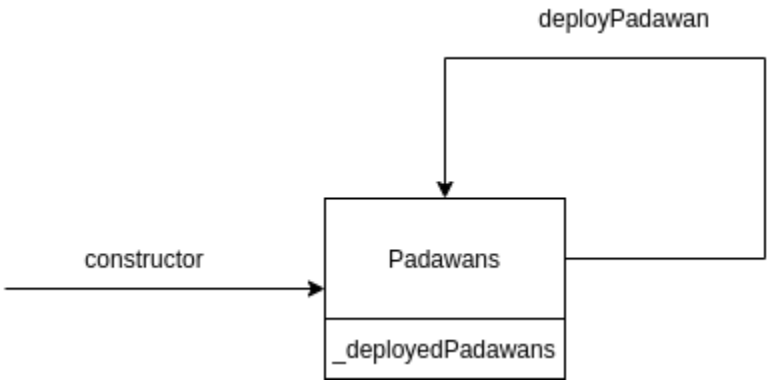


DM_padawans

This projection stays for the collection of deployed Padawans. Please note that it keeps only the list of `padawanId`s, rather than content and attributes of the Padawans themself.

Covers the following variables of the Demiurge contract:

- `_deployedPadawans`

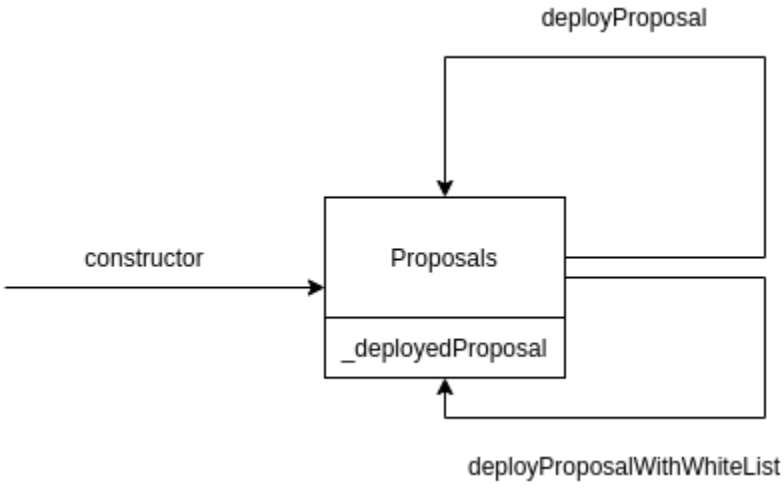


DM_proposals

This projection stays for the collection of deployed Proposals. Please note that it keeps only the list of `proposalId`s, rather than content and attributes of the Proposals themself.

Covers the following variables of the Demiurge contract:

- `_deployedProposals`

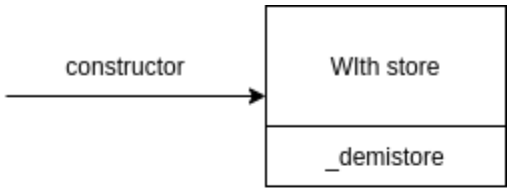


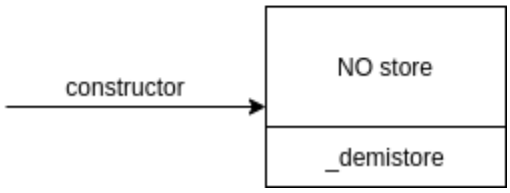
DM_store

This projection stays for the representation of the external *DemiurgeStore* if present. Keeps two states - with and without *DemiurgeStore*. As the changing of DemiurgeStore after construction is impossible (*onCodeUpdate* is considered separately) the movement between these two states does not exist.

Covers the following variables of the Demiurge contract:

- `demiStore`



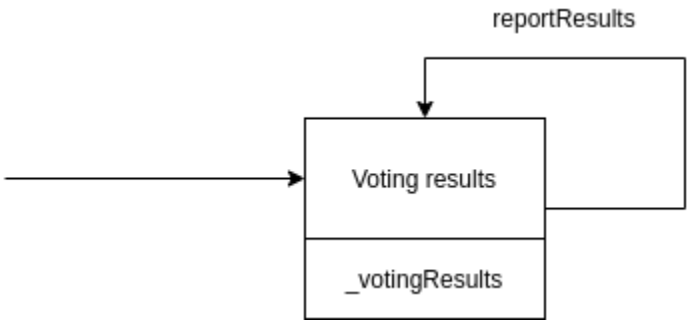


DM_voting

This projection stays for the collection of historical voting results.

Covers the following variables of the Demiurge contract:

- `_votingResults`

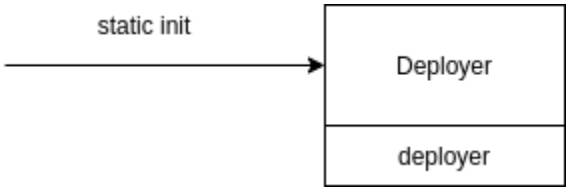


PW_deployer

This projection stays for a static member of Padawan called the deployer. This value can not be changed throughout the lifecycle.

Covers the following variables of the Padawan contract:

- `deployer`

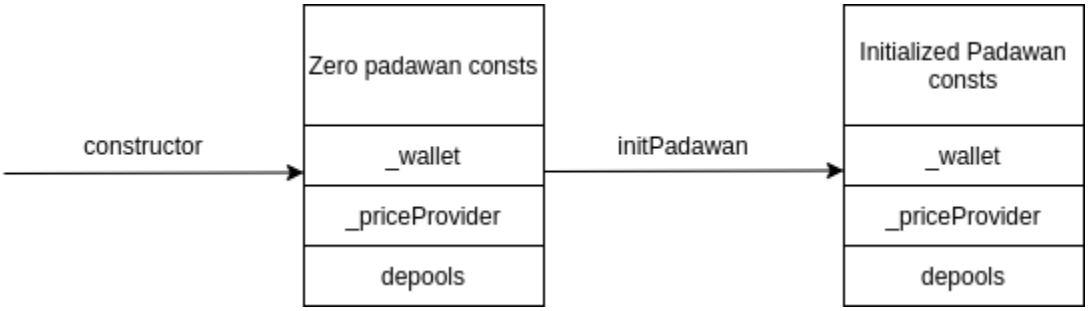


PW_const

This projection reflects those variables of Padawan contract that are considered as constant. So they are initially initialized by zeroes by the constructor, then really initialized by the *initPadawan* function and never updated afterwards.

Covers the following variables of the Padawan contract:

- `_wallet`
- `_priceProvider`
- `_depools`

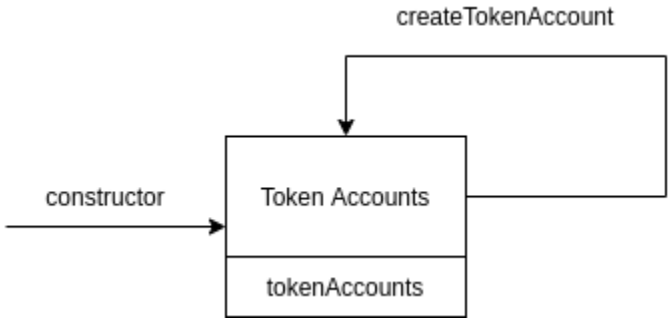


PW_tokenAccounts

This projection represents the collection of the token accounts used by a present Padawan. It is worth mentioning that it reflects exclusively the collections of root addresses without handling the content of each *TipAccount* object.

Covers the following variables of the Padawan contract:

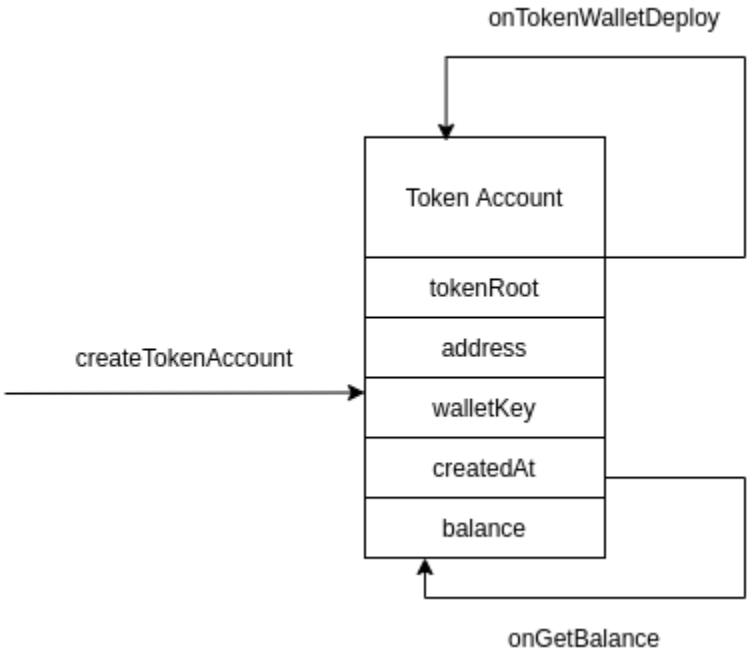
- `tokenAccounts`



PW_tokenAccount

This projection holds for a certain token account (identified by its token root) and all its fields.

It covers all the variables of a particular token account (element of `tokenAccounts`) of the Padawan contract:

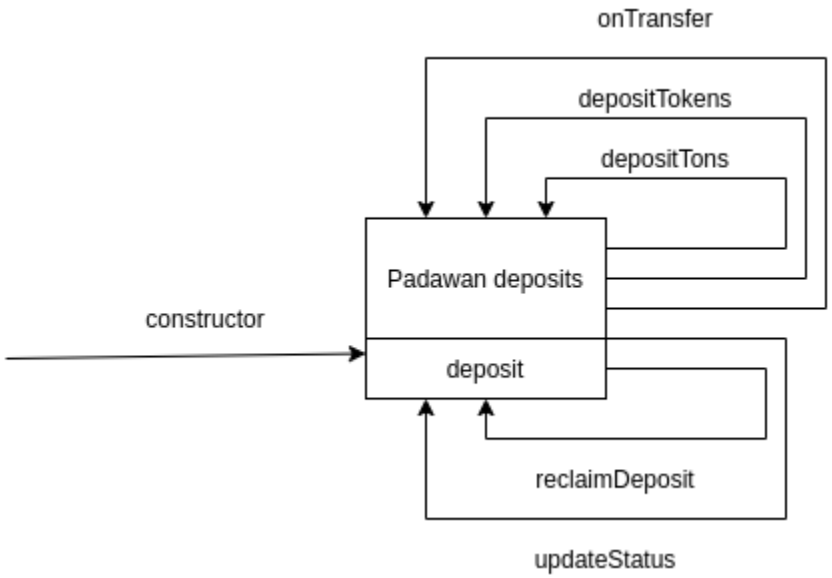


PW_deposits

This projection represents the collection of deposits. It is worth mentioning that it reflects exclusively collections of deposit ids without handling the content of each *Deposit* object.

Covers the following variables of the Padawan contract:

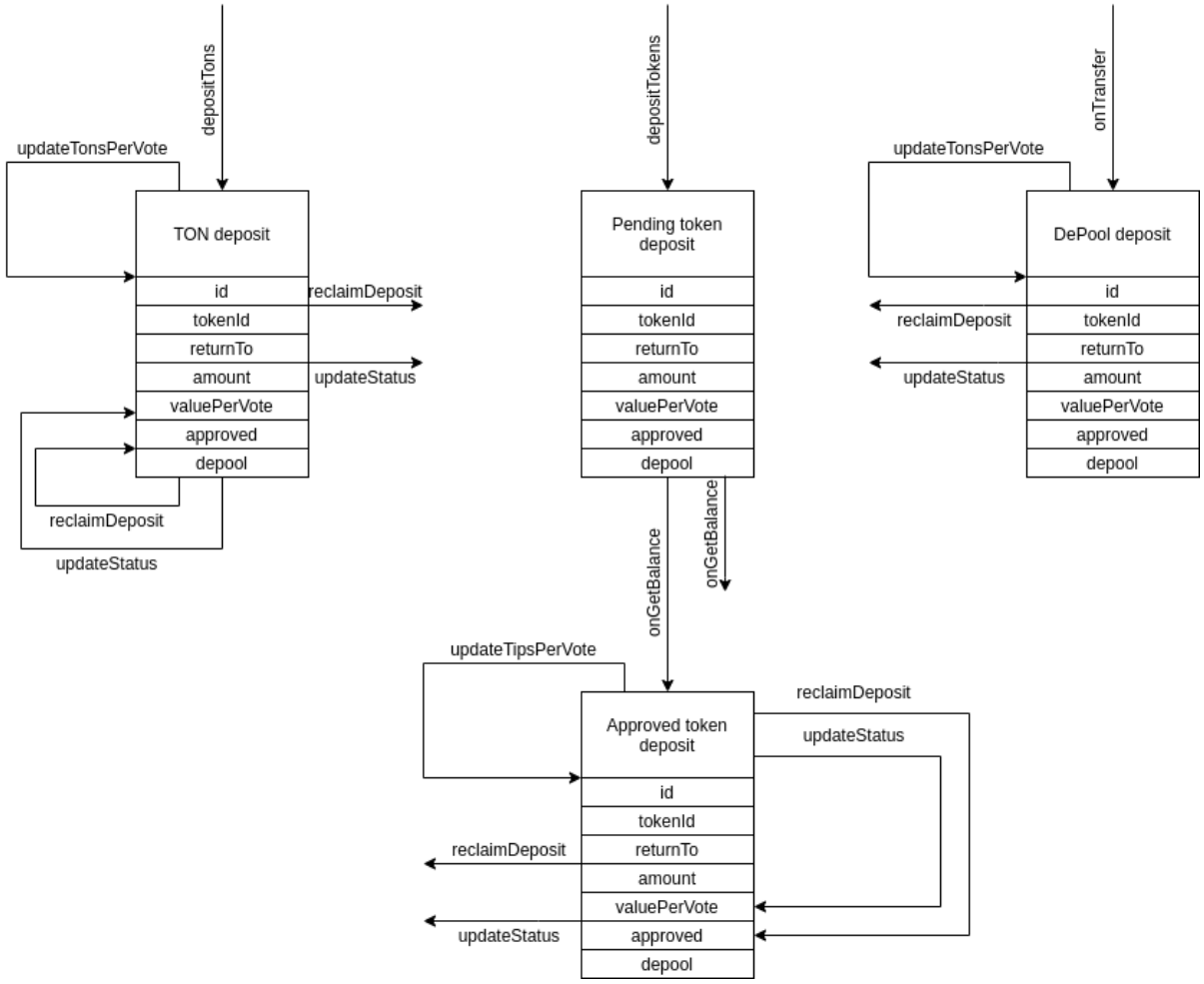
- `deposits`



PW_deposit

This projection covers the particular deposit of the Padawan contract and all its fields. The projection recognizes four different states - TON deposit, Pending token deposit, Token deposit and DePool deposit.

Covers all the variables of the particular deposit (element of `deposits`) of the Padawan contract:

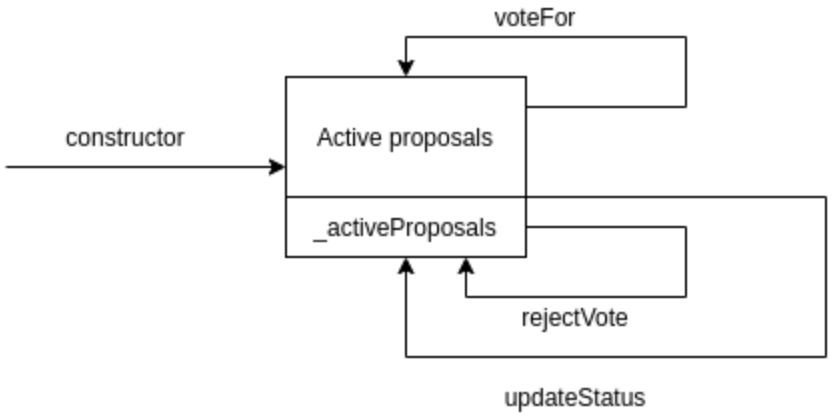


PW_proposals

This projection represents the collection of active proposals combined with votes assigned for each of them. It is worth mentioning that it reflects exclusively collections of root addresses without handling the votes spent for each proposal.

Covers the following variables of the Padawan contract:

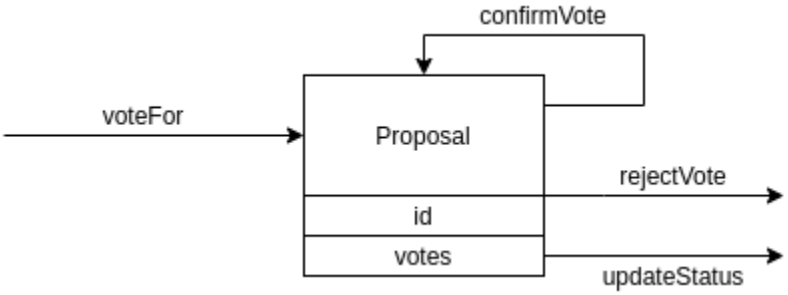
- `_activeProposals`



PW_proposal

This projection represents the certain Proposal as a combination of its address and number of votes provided by the present Padawan.

Covers address and votes for the certain Proposal:

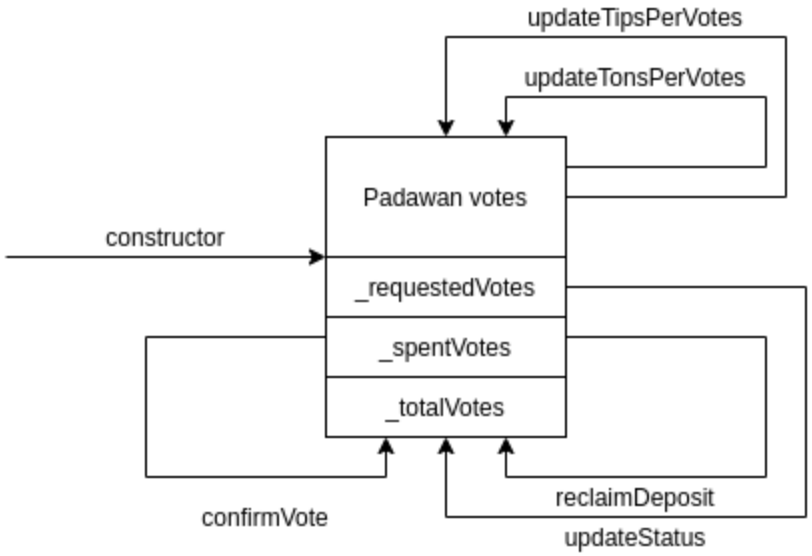


PW_votes

This projection reflects all the auxiliary variables of Padawan contract related to keeping the secondary voting information.

Covers the following variables of the Padawan contract:

- `_requestedVotes`
- `_totalVotes`
- `_spentVotes`

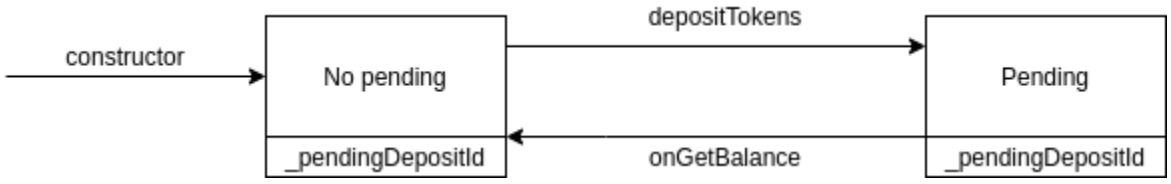


PW_pending

This projection reflects the state of the pending deposit of the Padawan contract. As each Padawan contract may have at most one pending deposit two states are distinguished: “no pending deposit” and “some pending deposit”.

Covers the following variables of the Padawan contract:

- `_pendingDepositId`

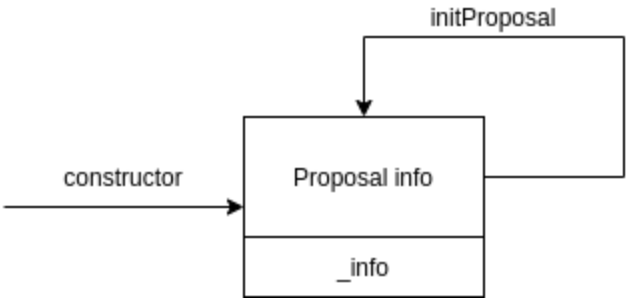


PP_info

This projection represents the state of *ProposalInfo* structure bound to the Proposal contract. Once initialized by the constructor as zero and later by *initProposal* it can not be changed afterwards.

Covers the following variables of the Proposal contract:

- `_info`

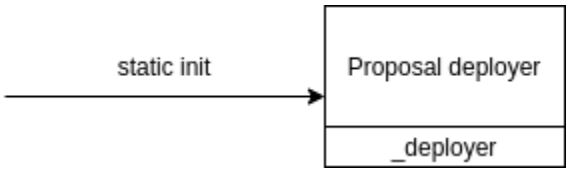


PP_deployer

This projection stays for a static member of Proposal called the deployer. This value can not be changed throughout the lifecycle.

Covers the following variables of the Proposal contract:

- `deployer`

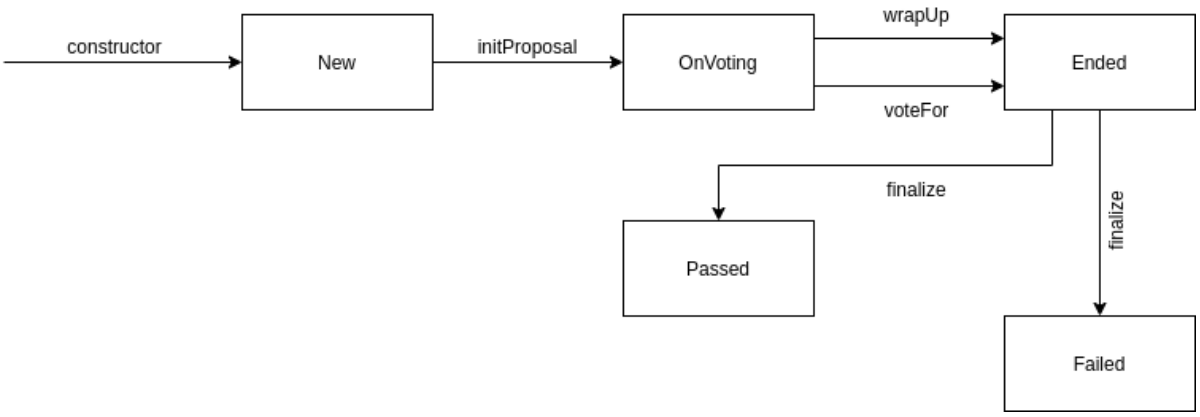


PP_state

This projection reflects the state of the Proposal. It has to be noted that while *ProposalState* structure supports a whole bunch of states only the subset of them is used by the SMV being specified.

Covers the following variables of the Proposal contract:

- *state* OF `_state`

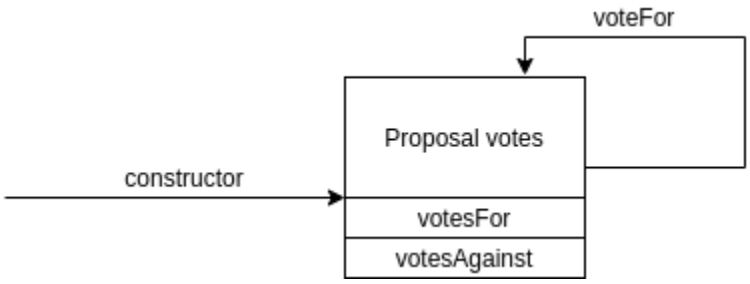


PP_votes

This projection reflects the votes spent for or against the Proposal.

Covers the following variables of the Proposal contract:

- *votesFor* OF `_state`
- *votesAgainst* OF `_state`

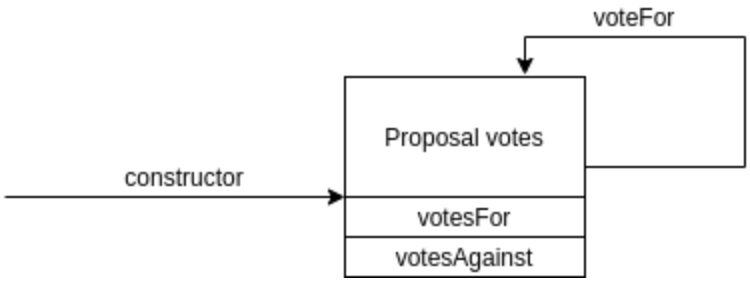


PP_constants

This projection represents the variables of Proposal contract that are supposed to be immutable after invoking *initProposal* function.

Covers the following variables of the Proposal contract:

- `_hasWhiteList`
- `_voters`
- `_padawanSI`
- `_votecountingmodel`

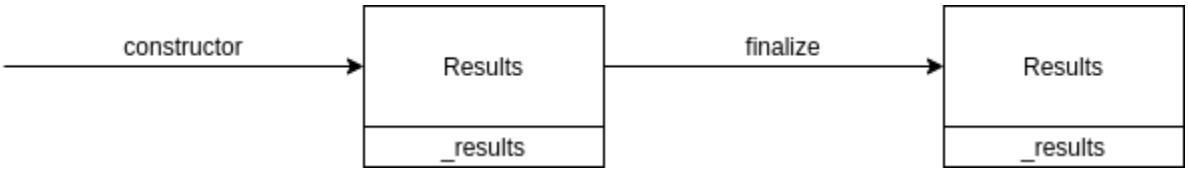


PP_results

This projection stays for voting results of the Proposal representing the *VotingResults* structure.

Covers the following variables of the Proposal contract:

- `_results`



User scenarios

The following business-level scenarios are suggested upon analysis of state machine projections described in the previous section. The basic rule is that each route for each diagram states for one scenario.

In case of cycles the single-loop reduced scenarios (that start from the machine entry point as well as “regular” scenarios but stop immediately after exiting the loop). These “loop” scenarios must cover all the possible loop branches but it’s never required to take the second loop.

Please note that scenarios based on projections are positive. There are also scenarios that are based on incorrect input data that should be rejected by the corresponding functions. Such scenarios may be automatically built by subsequent violation of the parameter requirements described in the “Specification” section and will not be discussed further in the present document.

The complete list of the projection-based scenarios is below:

Scenario I : Images with store

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Check `demiStore` variable
5. Check for CHECK_PASSED
6. Check *Padawan* image
7. Check *Proposal* image
8. Check price provider
9. Check depools

Scenario II : Images without store

1. Create *Demiurge* without *DemiurgeStore*
2. Check `demiStore` variable
3. Call in arbitrary order:
    a. *updateImage* (with *CHECK_PADAWAN*)
    b. *updateImage* (with *CHECK_PROPOSAL*)
    c. updateDepools
    d. updateAddress
4. Check for CHECK_PASSED
5. Check *Padawan* image
6. Check *Proposal* image
7. Check price provider
8. Check depools

Scenario III : Double setting of Padawan image

1. Create Demiurge without DemiurgeStore
2. Call *updateImage* (with *CHECK_PADAWAN* and some *Padawan* image)
3. Check *Padawan* image
4. Call *updateImage* (with *CHECK_PADAWAN* and another *Padawan* image)
5. Check *Padawan* image

Scenario IV : Double setting of Proposal image

1. Create Demiurge without DemiurgeStore
2. Call *updateImage* (with *CHECK_PROPOSAL* and some *Proposal* image)
3. Check *Proposal* image
4. Call *updateImage* (with *CHECK_PROPOSAL* and another *Proposal* image)
5. Check *Padawan* image

Scenario V : Double setting of depools

1. Create Demiurge without DemiurgeStore
2. Call *updateDepools* with some set of depools
3. Check depools
4. Call *updateDepools* with another set of depools
5. Check depools

Scenario VI : Double setting of price provider

1. Create Demiurge without DemiurgeStore
2. Call *updateAddress* with address of some price provider
3. Check price provider
4. Call *updateAddress* with address of another price provider
5. Check price provider

Scenario VII : Deploy padawan

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Check the list of deployed *Padawans*
5. Deploy *Padawan* from some *Wallet*
6. Check the list of deployed *Padawans*
7. Deploy *Padawan* from another *Wallet*
8. Check the list of deployed *Padawans*

Scenario VIII : Deploy proposal

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Check the list of deployed *Proposals*
5. Deploy some *Proposal*
6. Check the list of deployed *Proposals*
7. Deploy *Padawan* from some *Wallet*
8. Deploy white listed Proposal with the recently created *Padawan* in the white list
9. Check the list of deployed *Proposals*

Scenario IX : Voting results

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Deploy some *Proposal* with one vote required

6. Check `_votingResults`
7. Vote for *Proposal* with *Padawan*
8. Wait for voting end
9. Check `_votingResults`
10. Deploy another *Proposal* with one vote required
11. Wait for voting end
12. Check `_votingResults`

Scenario X : Padawan deployer

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Check `deployer`

Scenario XI : Padawan constants

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Check `_wallet`
7. Check `_priceProvider`
8. Check `depools`

Scenario XII : Token accounts

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Check `_tokenAccounts`
7. Create new token account from *Wallet*
8. Check `_tokenAccounts`
9. Create another token account from *Wallet*
10. Check `_tokenAccounts`

Scenario XIII : Token account

1. Create *DemiurgeStore* with all the attributes

2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Create new token account from *Wallet*
7. Wait for creation and initialization
8. Check values of recently created token account
9. Deposit some tokens using token account
10. Wait for deposit creation and initialization
11. Check values of token account
12. Deposit some more tokens using token account
13. Wait for deposit creation and initialization
14. Check values of token account

Scenario XIV : Padawan Deposits

1. Create *DemiurgeStore* with all the attributes
2. Ensure depools list is not empty
3. Create *Demiurge* with the previously created *DemiurgeStore*
4. Wait for creation and initialization
5. Deploy *Padawan* from some *Wallet*
6. Wait for creation and initialization
7. Deploy some *Proposal*
8. Wait for creation and initialization
9. Check deposits
10. Deposit some TONs
11. Check deposits
12. Reclaim deposits
13. Check deposits
14. Deposit some TONs
15. Vote for *Proposal*
16. Reclaim deposits
17. Check deposits
18. Wait for voting end
19. Check deposits
20. Deploy some *Proposal*
21. Wait for creation and initialization
22. Create new token account from *Wallet*
23. Wait for creation and initialization
24. Deposit some tokens
25. Wait for creation and initialization
26. Reclaim deposits
27. Check deposits
28. Deposit some tokens

29. Wait for creation and initialization
30. Vote for *Proposal*
31. Reclaim deposits
32. Check deposits
33. Wait for voting end
34. Check deposits
35. Deploy some *Proposal*
36. Wait for creation and initialization
37. Transfer some TONs from DePool
38. Wait for creation and initialization
39. Reclaim deposits
40. Check deposits
41. Transfer some TONs from DePool
42. Wait for creation and initialization
43. Vote for *Proposal*
44. Reclaim deposits
45. Check deposits
46. Wait for voting end
47. Check deposits

Scenario XV : TON deposit deleted when reclaimed

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Deposit some TONs
7. Wait for creation and initialization
8. Check deposit status
9. Reclaim deposit
10. Check the deposit has been deleted

Scenario XVI : TON deposit deleted when status updated

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Deploy some *Proposal*
7. Wait for creation and initialization
8. Deploy second Proposal
9. Wait for creation and initialization

10. Deposit some TONs
11. Wait for creation and initialization
12. Check deposit status
13. Vote for the first *Proposal*
14. Vote for the second *Proposal*
15. Reclaim deposit
16. Check deposit status
17. Wait for the first voting end
18. Check deposit status
19. Wait for the second voting end
20. Check the deposit has been deleted

Scenario XVII : Disapproved Token Deposit

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Create new token account from *Wallet*
7. Wait for creation and initialization
8. Deposit some tokens into *Padawan* (more tokens than available at the token account)
9. Wait for creation and initialization
10. Check the deposit has been deleted

Scenario XVIII : Token deposit deleted when reclaimed

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Create new token account from *Wallet*
7. Wait for creation and initialization
8. Deposit some tokens into *Padawan* (less tokens than available at the token account)
9. Wait for creation and initialization
10. Check deposit status
11. Reclaim deposit
12. Check the deposit has been deleted

Scenario XIX : Token deposit deleted when status updated

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*

3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Create new token account from *Wallet*
7. Wait for creation and initialization
8. Deposit some tokens into *Padawan* (less tokens than available at the token account)
9. Wait for creation and initialization
10. Check deposit status
11. Deploy some *Proposal*
12. Wait for creation and initialization
13. Deploy another *Proposal*
14. Vote for the first *Proposal*
15. Vote for the second *Proposal*
16. Wait for the confirmation of votes
17. Reclaim deposit
18. Check deposit status
19. Wait for the end of the first voting
20. Check deposit status
21. Wait for the end of the second voting
22. Check the deposit has been deleted

Scenario XX : DePool deposit deleted when reclaimed

1. Create *DemiurgeStore* with all the attributes
2. Ensure depools list is not empty
3. Create *Demiurge* with the previously created *DemiurgeStore*
4. Wait for creation and initialization
5. Deploy *Padawan* from some *Wallet*
6. Wait for creation and initialization
7. Transfer some TONs from *DePool*
8. Wait for creation and initialization
9. Check deposit status
10. Reclaim deposit
11. Check the deposit has been deleted

Scenario XXI : DePool deposit deleted when status updated

1. Create *DemiurgeStore* with all the attributes
2. Ensure depools are not empty
3. Create *Demiurge* with the previously created *DemiurgeStore*
4. Wait for creation and initialization
5. Deploy *Padawan* from some *Wallet*
6. Wait for creation and initialization
7. Deploy some *Proposal*

8. Wait for creation and initialization
9. Deploy second Proposal
10. Wait for creation and initialization
11. Transfer some TONs from DePool
12. Wait for creation and initialization
13. Check deposit status
14. Vote for the first *Proposal*
15. Vote for the second *Proposal*
16. Reclaim deposit
17. Check deposit status
18. Wait for the first voting end
19. Check deposit status
20. Wait for the second voting end
21. Check the deposit has been deleted

Scenario XXII : Padawan active proposals

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Check `_activeProposals`
7. Deploy some *Proposal*
8. Wait for creation and initialization
9. Vote for proposal with disallowed amount of votes
10. Check `_activeProposals`
11. Vote for proposal with allowed amount of votes
12. Check `_activeProposals`
13. Wait until voting ends
14. Check `_activeProposals`

Scenario XXIII : Padawan active proposal deleted when voting is rejected

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Deploy some *Proposal*
7. Wait for creation and initialization
8. Vote for proposal with disallowed amount of votes
9. Check the active proposal has been deleted

Scenario XXIV : Padawan active proposal deleted when status updated

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Deploy some *Proposal*
7. Wait for creation and initialization
8. Vote for proposal with allowed amount of votes
9. Check the proposal status
10. Wait for voting end
11. Check the active proposal has been deleted

Scenario XXV : Padawan votes

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Check `_requestedVotes`, `spentVotes` and `_totalVotes`
7. Deposit some TONs
8. Wait for deposit acceptance
9. Check `_requestedVotes`, `spentVotes` and `_totalVotes`
10. Create new token account from *Wallet*
11. Wait for creation and initialization
12. Deposit some tokens into *Padawan* (less tokens than available at the token account)
13. Check `_requestedVotes`, `spentVotes` and `_totalVotes`
14. Deploy some *Proposal*
15. Wait for creation and initialization
16. Vote for the *Proposal* with less than allowed amount of votes
17. Wait for voting acceptance
18. Check `_requestedVotes`, `spentVotes` and `_totalVotes`
19. Reclaim deposit
20. Check `_requestedVotes`, `spentVotes` and `_totalVotes`
21. Wait for voting end
22. Check `_requestedVotes`, `spentVotes` and `_totalVotes`

Scenario XXVI : Pending deposits

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*

3. Wait for creation and initialization
4. Deploy *Padawan* from some *Wallet*
5. Wait for creation and initialization
6. Check `_pendingDepositId`
7. Create new token account from *Wallet*
8. Immediately check `_pendingDepositId`
9. Wait for creation and initialization
10. Check `_pendingDepositId`

Scenario XXVII : Proposal info

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy some Proposal
5. Immediately check `_info`
6. Wait for creation and initialization
7. Check `_info`

Scenario XXVIII : Proposal deployer

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy some *Proposal*
5. Check `deployer`

Scenario XXIX : Proposal passed on time

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy some *Proposal*
5. Immediately check *state* OF `_state`
6. Wait for creation and initialization
7. Check *state* OF `_state`
8. Deploy a few *Padawans*
9. Wait for creation and initialization
10. Deposit some TONs to each *Padawan*
11. Vote for and against by some *Padawans* in that way where *Proposal* is passed on time
12. Wait until voting ends
13. Call *wrapUp* method
14. Immediately check *state* OF `_state`

15. Wait for result acceptance
16. Check *state* OF `_state`

Scenario XXX : Proposal failed on time

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy some *Proposal*
5. Immediately check *state* OF `_state`
6. Wait for creation and initialization
7. Check *state* OF `_state`
8. Deploy a few *Padawans*
9. Wait for creation and initialization
10. Deposit some TONs to each *Padawan*
11. Vote for and against by some *Padawans* in that way where *Proposal* is failed on time
12. Wait until voting ends
13. Call *wrapUp* method
14. Immediately check *state* OF `_state`
15. Wait for result acceptance
16. Check *state* OF `_state`

Scenario XXXI : Proposal passed ahead of time

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy some *Proposal*
5. Immediately check *state* OF `_state`
6. Wait for creation and initialization
7. Check *state* OF `_state`
8. Deploy a few *Padawans*
9. Wait for creation and initialization
10. Deposit some TONs to each *Padawan*
11. Vote for and against by some *Padawans* in that way where *Proposal* is passed ahead of time
12. Immediately check *state* OF `_state`
13. Wait for result acceptance
14. Check *state* OF `_state`

Scenario XXXII : Proposal failed ahead of time

1. Create *DemiurgeStore* with all the attributes

2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy some *Proposal*
5. Immediately check *state* OF `_state`
6. Wait for creation and initialization
7. Check *state* OF `_state`
8. Deploy a few *Padawans*
9. Wait for creation and initialization
10. Deposit some TONs to each *Padawan*
11. Vote for and against by some *Padawans* in that way where *Proposal* is failed ahead of time
12. Immediately check *state* OF `_state`
13. Wait for result acceptance
14. Check *state* OF `_state`

Scenario XXXIII : Proposal votes

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy some *Proposal*
5. Wait for creation and initialization
6. Check *votesFor* OF `_state` and *votesAgainst* OF `_state`
7. Deploy a *Padawan*
8. Wait for creation and initialization
9. Deploy another *Padawan*
10. Wait for creation and initialization
11. Deposit some TONs to each *Padawan*
12. Wait for deposit acceptance
13. Vote for the Proposal by the first Padawan
14. Check *votesFor* OF `_state` and *votesAgainst* OF `_state`
15. Vote against the Proposal by the second Padawan
16. Check *votesFor* OF `_state` and *votesAgainst* OF `_state`

Scenario XXXIV : Proposal constants

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy some *Proposal*
5. Immediately check for `_hasWhiteList`, `_voters`, `_padawanSI` and `_votecountmodel`
6. Wait for creation and initialization

7. Immediately check for `_hasWhiteList`, `_voters`, `_padawanSI` and `_votecountmodel`

Scenario XXXV : Proposal results

1. Create *DemiurgeStore* with all the attributes
2. Create *Demiurge* with the previously created *DemiurgeStore*
3. Wait for creation and initialization
4. Deploy some *Proposal*
5. Wait for creation and initialization
6. Check `_results`
7. Wait until voting ends
8. Check `_results`

Audit Results

Audit reveals several issues of different natures which can be ranged from minor to critical depending on point of view. We propose to make corresponding workarounds to be ready for production use. And based on commonly available information and developers negotiations the fixes have been made and may correspond to commit https://github.com/RSquad/dens-smv/commit/3a1c94610948e046aea063d14dc5f8c979c3793c (2021, May 31) which is out of current investigation due to time reason.

The audit results consist of two sections I and II.

I. The following observations has been made during the audit of the DENS version of SMV (which is later than originally proposed BFTG version) and corresponds to the commit https://github.com/RSquad/dens-smv/commit/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd (2021, May 17). This version of SMV contract is reduced in functionality compared with the original BFTG version, however as it has later development time for equivalent functions we suggest it is more actual to analyze it rather than original code.

1. _addrStore (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Demiurge.sol#L68) address can be zero after the constructor call, which is hardly a desirable results; still, it allows to call onlyStore functions by an external message which could lead to undesired behaviour
2. _addrTokenRoot (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Padawan.sol#L71) can be zero. We haven't discovered sufficiently critical behavior thus far, but in case of an expansion to a larger token mass, this is unlikely to

work because of reclaimDeposit function where zero address corresponds to `_crystalsID`.

3. _requestedVotes can be changed in the course of the first run of reclaimDeposit (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Padawan.sol#L121) . This doesn't lead to mistakes, but can be a source of undesired behavior. If there are two reclaimDeposit messages in the queue, unlockDeposit can start working with the value that came in the second one.
4. In fact, the field deposits[].approved (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Padawan.sol#L295) is not used in fact. Additionally, reclaimDeposit with `votes = 0` will touch this deposit (even though its status is not approved)
5. _pendingDepositID (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Padawan.sol#L301) field is critical, in the case of an uncompleted onGetBalance call (out of gas) it won't be reset to zero, ending up in a deadlock inside depositTokens (`require(_pendingDepositId == 0, Errors.PENDING_DEPOSIT_ALREADY_EXISTS);`).
6. The functions that call _unlockDeposit (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Padawan.sol#L185) or itself must perform a more complex balance analysis because it's a complicated one, and if the contract runs out of funds, it wouldn't change the state - the suggested solution is to perform tvm.commit after successful end of cycle.
7. reclaimDeposit parameter (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Padawan.sol#L121) `votes = 0` will run multiple useless cycles
8. For early completion purposes, `if (2 * no > _totalVotes)` ">" can be replaced with ">=" for more precise results (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Proposal.sol#L109)
9. We have to check the amount of internal messages sent in the functions that send them in the loop (e.g. reclaimDeposit (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Padawan.sol#L121)). We assume there cannot be more than 255 of them and therefore if not we have to make a workaround or just prevent the case when we need to send more messages than allowed in one TVM round.
10. We suggest reclaimDeposit can be **refactored**. In fact, there is a double loop - for proposals, and, within each proposal, for deposits (by way of updateStatus). We consider the functionality will not be impaired if we first go through all proposals, and then all deposits within just two loops.
11. There are some (technical) funds sent to e.g. Demiurge that cannot be taken out afterwards. The same should be checked for Proposal and Padawan as well.
12. The `updateCode` (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Demiurge.sol#L138) function can take **any** image

through its `code` argument, and therefore we can send there some arbitrary code (even malicious), although we could compare it with the hardcoded hash at an earlier stage (as simple workaround).

13. Old proposals stop working with new padawans if the latter are updated.
14. function `deployPadawan(address owner) external onlyContract {require(msg.value >= DEPLOY_FEE); // = 3.1 ton)...}` sends 5 TONs in fact (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Demiurge.sol#L84)
15. In the `vote` function (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Padawan.sol#L78) it is not checked whether we have enough funds for performing callbacks `confirmVote` or `rejectVote` - if we don't have them, we enter an inconsistent state. Moreover, if there's a potential **attack**, we can vote deliberately to have not enough funds and save the votes thereby as votes "withdraw" is performed in callbacks (and the proposal state has been already changed to the moment).
16. `ITokenWallet(acc.addr).transfer{value: 0.1 ton + 0.1 ton} (deposit.returnTo, value, 0.1 ton)` (https://github.com/RSquad/dens-smv/blob/c4825fb59d91d0d6f7dc8206193aed77e5bf13dd/src/Padawan.sol#L208) inside `unlockDeposit` doesn't update `acc.balance` balance, which leads to incorrect operation of `onGetBalance`. If we top up the balance (or, even simpler, if we set deposit.returnTo == acc.addr), the comparison balance >= dep.amount + prevBalance will fail.

II. The following observations correspond to the original BFTG version https://github.com/RSquad/BFTG/commit/b054d45d04ec9f04769b8e1e2e692b022234eb79 (2021, April 28) whether not overlapped with already mentioned issues in the section I.

1. We propose it is necessary to check that price <> 0 (not equal to zero) in the function _convertDepositToVotes (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L370). Furthermore, we consider it *counterintuitive* that this is not a price for a token in TONs but a price for a voice (which is assumed to be 1 TON priced) in tokens, and we have to *divide* by the price. Moreover, this value is always an integer, which implies that one TON (voice) cannot cost less than one token which is also out of common sense from our point of view.
2. If at `deposit.amount = math.min(value, deposit.amount)` in unlockDeposit (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L248) we find out that value = (deposit.amount/price)*price, the remainder of the division of deposit.amount/price will remain deposited and one can't take it out - in other words, the funds will be frozen.

3. The possibility of deposits less than 1 vote (currently 1 TON/vote) (actually one can create deposit with 0 parameterized `tons`) results in unused deposits which harms the system storage, and cannot be ever reclaimed (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L335 , https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L314)
4. _convertDepositToVotes (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L370): the price cannot be less than 1 Token per vote and PriceProvider cannot return, for example, 0, if this token is unknown to it, because division by 0 is impossible (it can return MAX_INT though). The latter will work however it makes the PriceProvider less intuitive as it should return “big” values for unknown tokens “inverted price”.
5. depositTokens checks whether newly created deposit id exists and fails if yes (which is correct), whereas onTransfer (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L359) doesn’t do so, and can mistakenly rewrite the existing deposit
6. The rationale for transferFunds (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L304) for Padawan is unclear. It is harmful as it doesn’t diminish the number of votes and removes funds, whereas in fact it is never called (we suggest however it is of technical purpose).
7. tvm.rawReserve (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L321) in depositTons is commented out, so the balance doesn’t remove in Padawan (we suggest however it is of debugging nature)
8. createTokenAccount (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L429) - it can create an account for an arbitrary contract that is not a real TIP3 contract, and the only way to know it has nothing is the PriceProvider contract (this is not a bug, but a considerable share of responsibility is placed on a contract that is not implemented at all)
9. unlockDeposit (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Padawan.sol#L248):
    - Doesn’t process transferStake errors
    - Doesn’t get the real transfer value but still uses it
    - The deposit information is erased, and the DePool reward is lost (alongside with everything that wasn’t transferred)
    - If the overall DePool stake win is less than minStake, we won’t transfer back the stake even though it’s free (not locked in the DePool round)

- If the DePool deposit has decreased (e.g. by Validator slashing), we are leaving the Padawan votes not backed by funds

see DePool implemenation at https://github.com/tonlabs/ton-labs-contracts/tree/master/solidity/depool

10. Current implementation of the interface for the PriceProvider interaction depends on the implementation of the latter - in case of a bad (or malicious) implementation of the PriceProvider this would break a Padawan's implementation (see updateTonsPerVote - it would call _convertDepositToVotes that could add non-existing votes). This is solvable at the Padawan level, whereby the same deposit would not be added twice.
11. SuperMajority 66% + 1 -> (3*no > total) ? true, false) can be corrected to 3 * no >= total (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Proposal.sol#L161)
12. Error in the formulas (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Proposal.sol#L130) For more information see section **"Soft majority and supermajority voting models"**
13. Very complex deploy chain (deployPadawan -> constructor -> onPadawanDeploy -> initPadawan)
14. _deployProposal (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Demiurge.sol#L232) that is run fast twice (before receiving the callback) shall "erase" the "first" proposal due to unchanged variable `_deployedProposalsCounter` which remains the same
15. initProposal (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Proposal.sol#L45): ProposalState.OnVoting might not correspond to its name (meaning that proposal is "on voting") when now < start
16. _calculateVotes (https://github.com/RSquad/BFTG/blob/b054d45d04ec9f04769b8e1e2e692b022234eb79/src/Proposal.sol#L130) must take into account number of the binary signs in arguments (e.g. the value `yes * total * 10` can lead to integer overflow). It must be revisited after main formulas reconsidered (see **"Soft majority and supermajority voting models").** Moreover `uint32 total` - the type is not sufficient as TIP3 provides for 2^128 -1 tokens as max supply.

Key security and reliability threats

The key security and reliability threats are:

- Unauthorized actions - ability to call contract functions by parties that are not entitled to **(critical)**

- Code fraud - attempt to use a frauded code cell thus violating the normal business login of the Token contract (**critical**)
- Emergency fund freezing - occurs when contract execution stops in the middle due lack of balance or any kind of exception some funds are frozen in the intermediate place and never can be released from there **(severe)**
- Improper fund distribution - incorrect fee, incorrect amounts, sending to the wrong recipient, any kind of intentional or accidental violation of the proper behavior **(critical)**
- Limits overflow - occurs when too many actions of a particular kind take place within one temporal (real or virtual) interval, more than allowed by the system itself and so fails to complete the action **(major)**
- Gas exhaustion - too much gas was used for the TVM execution, unexpected exception occurred **(from major to critical)**
- Replay attack - attempt to repeat the intercepted message literally, without even need to decode it to get it invoked multiple times (**critical**)
- Moving to incorrect state **(from major to critical)** - depending on ability to return to the correct state
- Not coming to eventual correctness **(from major to critical)** - eventual correctness assumes the state where the system comes at some point. If it does not happen it's a bug that may be even critical
- Regular bugs **(from minor to critical)**

Also should be noted that the list above covers exclusively threats internal to the *SMV* ecosystem itself while such external threats as private key stealing, blockchain or TVM malfunction.

The formal verification efforts provided at the later stages of verification will eliminate the risk of some most important types of threats such as correctness bugs, resource exhausting bugs, limits overflow, violations of the specification.