TIP-3 contract verification report (Stage1)

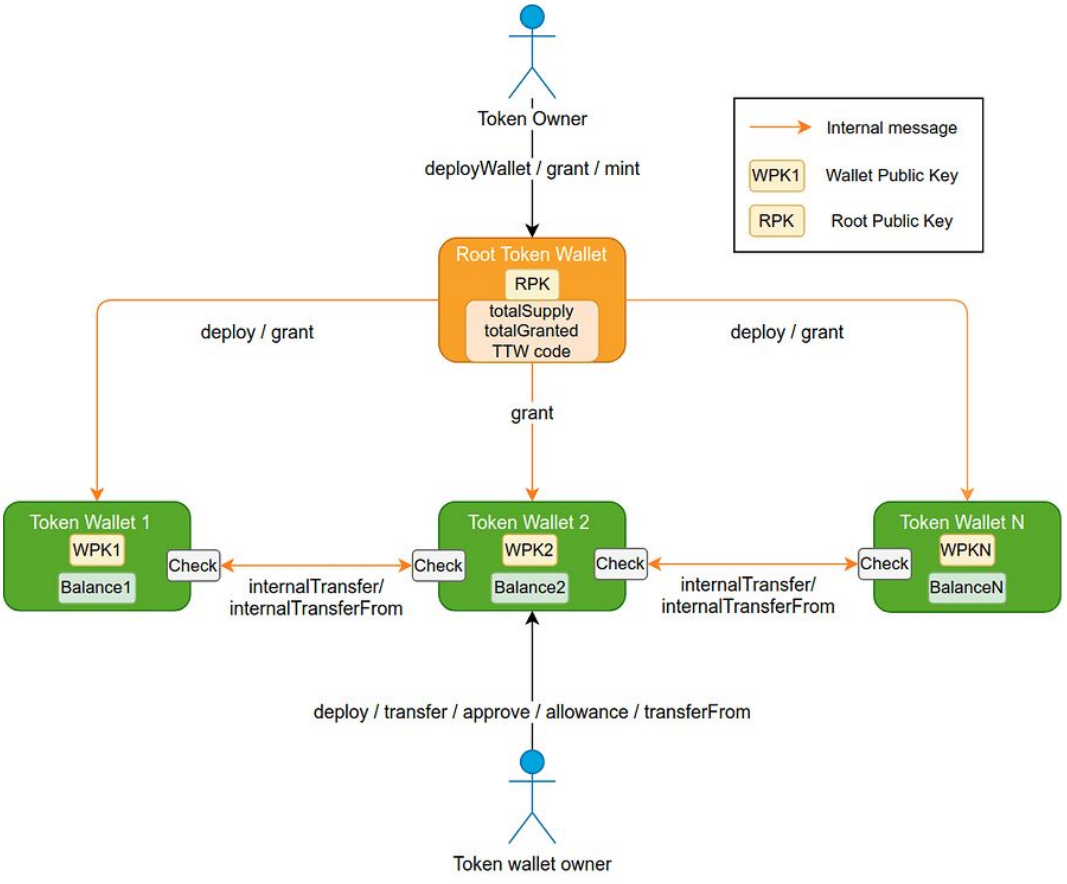
1. The purpose of the present document

The present document describes high level approach, methodology and business-level scenarios for the formal verification of the Free TON TIP-3 smart contract. Its mission is to provide the detailed description for items mentioned below:

- Formal description of the contact behaviour in the different cases
- Possible malfunctions and attacks
- Approaches how to formally verify (prove) that the behavior is exactly the same as described in this articles and that suspected attacks are not possible

2. Brief description of TIP-3 contract

TIP-3 contract allows users to create their own tokens. Unlike *ERC20*[1] in *Ethereum* the contract (called as the “Root” contract) does not keep all the balances inside (does not maintain ledger) but rather creates an individual “Token” contract for each wallet. The interaction between “Root” and “Token” contracts is illustrated by the following picture provided by the contest organizers.



[1] https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

Both types of contracts have the API provided by the contest organizers. This API as well as informal specification provided in the natural language may be obtained here[2].

3. The key underlying solutions

The proposed solution is based on *Coq Proof Assistant*[3]. This tool is primarily designed to make an environment for proving mathematical theorems and for this purpose it provides *OCaml*[4]-like language (named *Gallina*) for specifying the entities to be proved, specific language for proves (that is roughly the sequence of so called tactics (that stand for a step in terms of traditional proving[5]) as well as a specific language *Itac* for defining custom tactics.

In addition *Coq* provides its own comprehensive IDE as well as API to be integrated with other development environments such as *Microsoft Visual Studio Code*.

Coq itself is based on a pure mathematical paradigm called *Calculus of constructions*[6] (and its extension called *Calculus of Inductive Constructions*[7]) that allows to use mathematical induction[8] in addition to pure formal logic[9].

Coq was initially introduced in 1989, was dramatically developed since that time, used for many theoretical and practical applications and, as an outcome, the authors of the present document suggest to consider it as a reliable tool that means:

- If *Coq* states that some statement is proved it's considered as proved
- At the same time *Coq* may have any number of bugs not related to the statement written above

According to *Curry–Howard isomorphism*[10] all the mathematical proofs can be applied to the computer programs that is essential for the approach presented in the current document. Thus this proof assistant may be applied to the computer programs using the approaches described in the next section.

4. High-level approaches and methodologies

---

[2] https://forum.freeton.org/t/tip-3-distributed-token-or-ton-cash/64
[3] https://coq.inria.fr/
[4] https://ocaml.org/
[5] For example, ***apply*** tactic roughly means usage of some already known theorem or ***symmetry*** tactic utilizes the axiom that ***a=b*** is equivalent to ***b=a***
[6] https://hal.inria.fr/file/index/docid/76024/filename/RR-0530.pdf
[7] https://coq.inria.fr/distrib/current/refman/language/cic.html
[8] https://encyclopediaofmath.org/index.php?title=Mathematical_induction
[9] http://www.collegepublications.co.uk/logic/mlf/?00029
[10] 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.

As it was mentioned in the previous section *Coq* uses *OCaml*-like *Gallina* pure functional language while the smart contract to be verified is written using *C++* language[11] that is imperative.

To convert *C++* code into *Gallina* the following stuff is already implemented:

- *Gallina*-level *C++* DSL[12]
- Translator from *C++* to *Gallina* DSL

As a result the primitive *C++* operations are explicitly described and, if necessary, proven using Coq, while some limitations are still in place. The main of them is that all the cycles must have strong evidence of breaking at some point. This limitation is fundamental and is based on the *Turing halting problem*[13]. To avoid it developers are highly encouraged to use a weak normalization (or roughly, with decreasing explicit iterator variable). Otherwise, the conversion of such a code ceases to be automatic and requires significant (and non-trivial) manual efforts to turn the *C++* code into one acceptable by *Coq* (however, it's still doable).

The resulting behavior *C++* DSL primitives is implemented by a deep embedding technique while their imperative interaction by each other is reached by a specially designed monadic[14] model that lets one to implement imperative behavior in a pure functional language.

As an outcome any features of the inspected *C++* code may be proven using the DSL conversion described above with help of *Coq Proof Assistant* as well with usage of the specially designed libraries called *Finproof Base* (already implemented).

The approach described above allows ones to formally verify the *C++* code itself assuming that all the underlying tools work correctly. However, while it's assumed that *TON Virtual Machine*[15] and TON Blockchain[16] itself works correctly the compiler still may be considered as unreliable (upon community request).

While the full formal verification of the compiler is considered beyond the scope of this project it may be verified using the already developed *Coq-based TON Reference Virtual Machine (CTRVM)*. This virtual machine was developed strictly according to the *TON Virtual Machine* specification, and has proofs for the most of its parts (some parts, as hashmaps are still in progress but don't prevent the machine from usage).

A semi-automated runner may run the contract being inspected and ensure that the compiled TVM code works exactly in the manner as the *C++* one.

---

[11] While the original Solidity was developed exclusively for Ethereum the TON-adopted version is considered here

[12] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. ACM Computing Surveys, 37(4):316–344, 2005.doi:10.1145/1118890.1118892

[13] 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

[14] Wadler, Philip (June 1990). Comprehending Monads. ACM Conference on LISP and Functional Programming. Nice, France. CiteSeerX 10.1.1.33.5381

[15] https://test.ton.org/tvm.pdf

[16] https://test.ton.org/tblkch.pdf

As an outcome not only the high-level *C++* code is formally verified but also the compiler with limitations to its usage exclusively for the contract being inspected.

5. Pre Assumptions

The following presumptions are suggested for the verification of the present smart contract:

- Blockchain and TVM work strictly according to the specifications mentioned above
- Coq Proof Assistant or any other tool being used works correctly
- If something is not specified the assumptions based on common sense logic are applied
- Infinite sequences of similar elements (arrays) may exist
- Each cycle must exit at some precalculated point
- No direct or indirect recursions (function-based or message-based) allowed

6. Roles and responsibilities

The following roles may be identified for the TIP-3 contract:

- Root owner - the owner and the maintainer of the whole token ecosystem, the only person who can mint tokens and distribute initial supply. Also can create new non-empty wallets via Token contract deployment
- Token owner - the owner of the particular wallet (Token contract), may create a new wallet, accept grants from root contract, transfer tokens to any other contracts, approve and disapprove allowance
- Token receiver - subrole of a Token owner whose wallet can receive tokens from other Token contracts (with the same *name*, *symbol*, *decimals*, *code* and *root*)
- Token spender - subrole of Token owner whose wallet received allowance from some other Token contract. This role includes ability to get the allowed number of tokens in favor or itself or third-party
- Generic user - may just call getters to get different kinds of information about root contract or particular wallet

7. State of TIP-3 contract

The full state of the TIP-3 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
`name`	Unique name of the root contract
`symbol`	Unique symbol (such as BTC) of the root contract

`decimals`			Decimal logarithm of division of the base tokens to “displayable” tokens of the same value
`public_key`			Owner public key
`wallet_code`			Code of the Token contract as cell[17]
`total_supply`			The overall number of tokens in the system
`granted_supply`			The overall number of tokens granted to the Token contracts
`owner`			The address who created the Root contract
`token_contracts`			List of Token contracts attached to this Root contract (**please note it’s a virtual list that may not physically exist**)
	`workchain_id`		Workchain id for the Token contract
	`root_name`		Name of the Root contract
	`root_symbol`		Symbol of the Root contract
	`root_decimals`		Decimals of the Root contract
	`root_public_key`		Public key of the Root contract
	`wallet_public_key`		Public key of the Token contract
	`root_address`		Address of the Root contract
	`balance`		Balance of the token contract
	`token_address`		Address of the token contract
	`token_owner`		Address of the token owner
	`allowance_info`		Structure the keeps allowance information
		`spender`	Address allowed to spend tokens
		`remaining_tokens`	Amount of tokens still allowed to spend
`world`			All the entities outside the given Root contract and the underlying Token contracts
	`world_names`		List of the root contract names other than given one
	`world_symbols`		List of the root contract symbols other than given one

[17] Cells are the base data objects of TVM and well described by the TVM specification (https://test.ton.org/tvm.pdf)

	`world_public_keys`	List of the all public keys used in the blockchain other than public keys of the given Root contract and the underlying Token contracts

8. Structure of cell and slices specific for TIP-3 contract

The TIP-3 contract uses one specific cell - the wallet code cell and one specific slice[18] - used for *onBounce* messages. Each of them is discussed in the present section.

The cell with code must be identical to the code generated during compilation of the formally verified Token contract otherwise fraud is possible. Also before sending deploying the Token contract the Root contract must include a “boot” section to eliminate the risk of replay attack. In case of deploying the Token contract by constructor the “boot” section must be already included into the code. One more limitation for the Token contract code is that it can not have such instructions as *SETCODE*, *SETLIBCODE* or *CHANGELIB*.

The correct Token contract to be deployed must be a conjunction of:

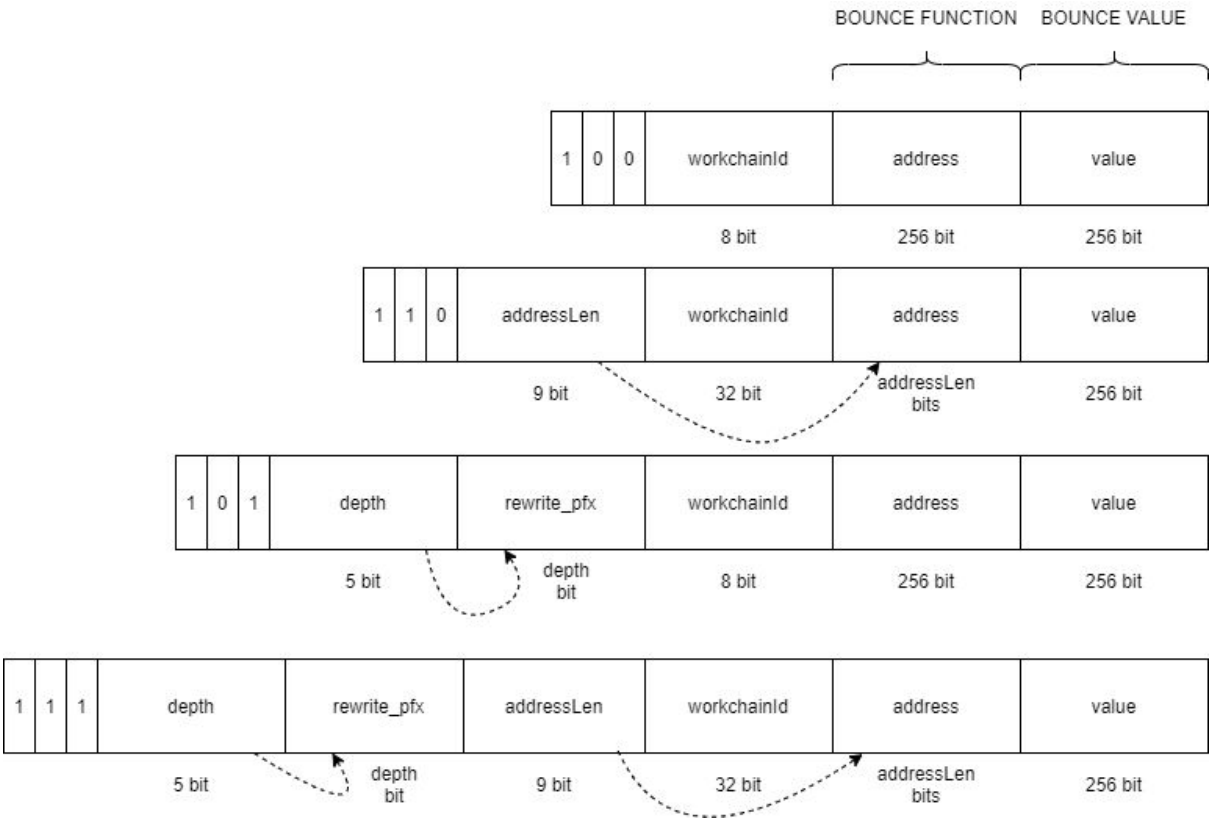
- name
- symbol
- decimals
- balance (*0* for *deployWallet*)
- root_public_key
- wallet_public_key
- root_address
- owner_address
- wallet_code (with “boot” section described above)
- allowance (must be zero for contract deployment)
- workchain_id

All the data mentioned above is encoded as a cell, while wallet_code is equivalent to the code of the Token smart contract.

The address of the Token contract must be deployed from the hash of all the data mentioned above. Any other ways to make a unique and non-reproducible address from the list above are considered as correct.

msg_body slice used for handling bouncing of internal message must have the following structure to be considered as correct:

[18] Slices are mutable objects designed to read cell content chunk by chunk (to some extent they are close to input streams in traditional languages). For more information refer to TVM documentation



For further details please refer to the [TVM specification].

9. Specification of TIP-3 contract in the formalized natural language

Here we provide the specification of the *TIP-3* contract in the formalized natural language that means each expression may be literally translated (presumably, manually) to the formal computer language (such as *Coq*[19] described above). All the state elements (described in the previous article) are underscored while the parameters are italic. The following meanings of English words (when capitalized) are used:

- 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 - mandatory requirement, may be translated as an ∀ quantifier for the left side to the right
- EMPTY - list with no elements
- EQUAL(S) - 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 sequence of functions) being described EQUALS to the corresponding projection before the call
- NO CHANGES OCCUR - the state is UNCHANGED (but gas, balance or any other non-contract specific states if they considered as a part of the state)
- ARE NOT MET - implication from the negation of the left part to the right part

[19] https://coq.inria.fr/

- ZERO - integer zero
- UNIQUE IN … IN TERMS OF - No element of mapping[20] 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
- PROPER CODE CELL - a cell that is a correct code cell as described in the section above
- BOUNCE FUNCTION - the function that is encoded in the bounce message body as described in the section above
- BOUNCE VALUE - the value that is encoded in the bounce message body as described in the section above
- PROPER BOUNCE SLICE - the slice is the correct bounce message slice as described in the section above
- CORRECT STATE INFO WITH - the correct StateInfo as described in the section above

1. Function-level specification
    a. Root contract functions
        i. constructor
            1. Access
                a. NO access restrictions
            2. Parameters
                a. name
                    i. NO constraints
                b. symbol
                    i. NO constraints
                c. decimals
                    i. *decimals* MUST be NON NEGATIVE

[20] By *mapping* the invocation of list-wide transformation often named as *map* (in such languages as *Haskell* or *Java*(*java.util.stream.map*)) is considered

ii. *decimals* MUST be LESS OR EQUAL 255

d. root_public_key

i. *root_public_key* MUST be POSITIVE

e. wallet_code

i. wallet_code MUST be a PROPER CODE CELL

f. total_supply

i. *total_supply* MUST be POSITIVE

3. Output
    a. name EQUALS to *name*
    b. symbol EQUALS to *symbol*
    c. decimals EQUAL to *decimals*
    d. public_key EQUALS to *root_public_key*
    e. wallet_code EQUALS to *wallet_code*
    f. total_supply EQUALS to total_supply
    g. token_contracts MUST be EMPTY
    h. world is UNCHANGED
    i. granted_tokens EQUAL to ZERO
    j. owner MUST be the Caller
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

ii. deployWallet

1. Access
    a. Caller MUST be an owner
2. Parameters

a. workchain_id

i. *workchain_id* MUST be NON NEGATIVE

b. pub_key

i. *pub_key* MUST be POSITIVE

ii. pub_key MUST be UNIQUE IN token_contracts IN TERMS OF wallet_public_key AND workchain_id

c. tokens

i. *tokens* MUST be POSITIVE

ii. *tokens* MUST be LESS OR EQUAL than total_supply MINUS granted_supply

d. grams

i. grams MUST be POSITIVE

ii. grams MUST be LESS OR EQUAL than the Root contract balance

3. Output
    a. name is UNCHANGED
    b. symbol is UNCHANGED
    c. decimals are UNCHANGED
    d. root_public_key is UNCHANGED
    e. wallet_code is UNCHANGED
    f. total_supply is UNCHANGED

g. token_contracts MUST be BEING ADDED WITH **new_contract** where **new_contract** has the following attributes:
    i. workchain_id EQUALS to *workchain_id*
    ii. root_name EQUALS to name
    iii. root_symbol EQUALS to symbol
    iv. root_decimals EQUAL to decimals
    v. root_public_key EQUALS to public_key
    vi. wallet_public_key EQUALS to *pub_key*
    vii. root_address EQUALS to “My Address”
    viii. balance EQUALS to *tokens*
    ix. token_address is the address of the **new_contract** as described in the section above
    x. spender EQUALS to ZERO
    xi. remaining_tokens EQUAL to ZERO
    xii. token_owner EQUALS to internal_owner
h. world is UNCHANGED
i. granted_tokens INCREASED BY *tokens*
j. owner is UNCHANGED

4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR
5. Return value
    a. Address of the **new_contract**
6. Balances
    a. *grams* MUST be LESS OR EQUAL than contract balance
    b. IN CASE OF SUCCESS the contract balance is deducted by *grams* AND the balance of new contract is increased by grams MINUS fee OTHERWISE balances are UNCHANGED (but fee)

iii. grant

1. Access
    a. Caller MUST be an owner
2. Parameters
    a. address
        i. MUST EXIST in token_contracts IN TERMS OF token_address (**token_contract**)
    b. tokens
        i. *tokens* MUST be POSITIVE
        ii. *tokens* MUST be LESS OR EQUAL than total_supply MINUS granted_supply
    c. grams
        i. grams MUST be POSITIVE
        ii. grams MUST be LESS OR EQUAL than the Root contract balance
3. Output

a. name is UNCHANGED
b. symbol is UNCHANGED
c. decimals are UNCHANGED
d. root_public_key is UNCHANGED
e. wallet_code is UNCHANGED
f. total_supply is UNCHANGED
g. token_contracts are UNCHANGED BUT the contract that EXIST in token_contracts IN TERMS OF token_address (**token_contract**) that has all the values UNCHANGED but balance INCREASED BY *tokens*
h. world is UNCHANGED
i. granted_tokens INCREASED BY *tokens*
j. owner is UNCHANGED

4. Exceptions
   a. If the constraints for the parameters ARE NOT MET
   b. exception must be raised and NO CHANGES OCCUR
5. Balances
   a. *grams* MUST be LESS OR EQUAL than balance
   b. IN CASE OF SUCCESS the balance is deducted by *grams* AND the balance of new contract is increased by grams MINUS fee OTHERWISE balances are UNCHANGED (but fee) UNCHANGED

iv. mint

1. Access
   a. Caller MUST be an owner
2. Parameters
   a. tokens
      i. *tokens* MUST be POSITIVE
3. Output
   a. total_supply is increased by *tokens*
   b. All the other state variables are UNCHANGED
4. Exceptions
   a. If the constraints for the parameters ARE NOT MET
   b. exception must be raised and NO CHANGES OCCUR

v. getName

1. Access
   a. NO access restrictions
2. Return value
   a. name

vi. getSymbols

1. Access
   a. NO access restrictions
2. Return value
   a. symbol

vii. getDecimals

1. Access
   a. NO access restrictions
2. Return value

a. decimals

viii. *getRootKey*
1. Access
   a. NO access restrictions
2. Return value
   a. public_key

ix. *getTotalSupply*
1. Access
   a. NO access restrictions
2. Return value
   a. total supply

x. *getTotalGranted*
1. Access
   a. NO access restrictions
2. Return value
   a. total granted

xi. *getWalletCode*
1. Access
   a. NO access restrictions
2. Return value
   a. wallet_code

xii. *getWalletAddress*
1. Access
   a. NO access restrictions
2. Parameters
   a. *workchain_id*
      i. *workchain_id* MUST be NON NEGATIVE
   b. *pub_key*
      i. *pub_key* MUST EXIST IN token_contracts IN TERMS OF workchain_id AND wallet_public_key
3. Exceptions
   a. If the constraints for the parameters ARE NOT MET
   b. exception must be raised and NO CHANGES OCCUR
4. Return value
   a. wallet_public_key OF EXISTING token_contracts IN TERMS OF workchain_id AND wallet_public_key

xiii. *onBounce*
1. Access
   a. Called by the system only in case of failure to deliver message
2. Parameters
   a. *msg*
      i. NO constraints
   b. *msg_body*
      i. *msg_body* MUST be a PROPER BOUNCE SLICE

ii. BOUNCED FUNCTION from *msg_body* MUST be *accept*
iii. BOUNCED VALUE from *msg_body* MUST BE LESS OR EQUAL to total_granted

3. Output
    a. total_granted is DECREASED BY BOUNCED VALUE from *msg_body*
    b. All other fields ARE UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

xiv. fallback

1. Access
    a. Called as a handler for all the internal messages other than explicitly mentioned above
2. Parameters
    a. All the parameters have NO constraints
3. Output
    a. All the fields are UNCHANGED

b. Token contract functions (**NOTE!!! For this section all the values are restricted to the Token contract hypersurface unless another explicitly stated**)

i. constructor

1. Access
    a. NO access restrictions
2. Parameters
    a. name
        i. *name* EQUALS to name
    b. symbol
        i. *symbol* EQUALS to symbol
    c. decimals
        i. *decimals* EQUAL to decimals
    d. root_public_key
        i. *root_public_key* EQUALS to public_key
    e. wallet_public_key
        i. *wallet_public_key* is POSITIVE
    f. root_address
        i. *root_address* EQUALS to root_address
    g. code
        i. *code* MUST be a PROPER CODE CELL
3. Output
    a. workchain_id EQUALS to id of the current workchain
    b. root_name EQUALS to *name*
    c. root_symbol EQUALS to *symbol*
    d. root_decimals EQUALS to *decimals*
    e. root_public_key EQUALS to *root_public_key*
    f. wallet_public_key EQUALS to *wallet_public_key*

g. root_address EQUALS to *root_address*
h. balance EQUALS to ZERO
i. token_address EQUALS to “My Address”
j. spender EQUALS to ZERO
k. remaining_tokens EQUAL to ZERO
l. token_owner EQUALS to Caller

4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

ii. transfer

1. Access
    a. Caller MUST be a token_owner
2. Parameters
    a. dest
        i. *dest* MUST be a valid address
    b. tokens
        i. *tokens* MUST be POSITIVE
        ii. *tokens* MUST be LESS OR EQUAL than balance
    c. grams
        i. grams MUST be POSITIVE
        ii. grams MUST be LESS OR EQUAL than the Token contract balance
3. Output
    a. balance IS DECREASED BY *tokens*
    b. All the other fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR
5. Balances
    a. *grams* MUST be LESS OR EQUAL than Token contract balance
    b. IN CASE OF SUCCESS the Sender contract balance is deducted by *grams* AND the balance of the Receiver contract balance is increased by grams MINUS fee OTHERWISE balances are UNCHANGED (but fee) UNCHANGED

iii. accept

1. Access
    a. Caller MUST be EQUAL to root_address
2. Parameters
    a. tokens
        i. *tokens* MUST be POSITIVE
3. Output
    a. balance MUST be INCREASED BY *tokens*
    b. Other fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET

b. exception must be raised and NO CHANGES OCCUR

iv. *internalTransfer*
1. Access
   a. Caller MUST EXIST IN token_contract IN TERMS OF token_address
2. Parameters
   a. *senderKey*
      i. *senderKey* MUST EXIST IN token_contract IN TERMS OF wallet_public_key
   b. *tokens*
      i. *tokens* MUST be POSITIVE
3. Output
   a. balance MUST be INCREASED BY *tokens*
   b. Other fields are UNCHANGED
4. Exceptions
   a. If the constraints for the parameters ARE NOT MET
   b. exception must be raised and NO CHANGES OCCUR

v. *getName*
1. Access
   a. NO access restrictions
2. Return value
   a. name

vi. getSymbols
1. Access
   a. NO access restrictions
2. Return value
   a. symbol

vii. *getDecimals*
1. Access
   a. NO access restrictions
2. Return value
   a. decimals

viii. *getBalance*
1. Access
   a. NO access restrictions
2. Return value
   a. balance

ix. *getWalletKey*
1. Access
   a. NO access restrictions
2. Return value
   a. wallet_public_key

x. *getRootAddress*
1. Access
   a. NO access restrictions
2. Return value
   a. root_address

xi. *allowance*

1. Access
    a. NO access restrictions
2. Return value
    a. allowance_info

xii. approve

1. Access
    a. Caller MUST be a token_owner
2. Parameters
    a. spender
        i. *spender* MUST be a valid address
    b. remainingTokens
        i. *remainingTokens* MUST be NON NEGATIVE
    c. tokens
        i. *tokens* MUST be POSITIVE
3. Output
    a. spender EQUALS to *spender*
    b. IF remaining_tokens EQUAL *remainingTokens* THEN remaining_tokens EQUALS to *tokens* OTHERWISE remaining_tokens are UNCHANGED
    c. Other fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

xiii. transferFrom

1. Access
    a. Caller MUST be a Token Owner
2. Parameters
    a. dest
        i. *dest* MUST be a valid address
    b. to
        i. *to* MUST be a valid address
    c. tokens
        i. *tokens* MUST be POSITIVE
    d. grams
        i. *grams* MUST be POSITIVE
3. Output
    a. All the fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR
5. Balances
    a. Balance of the Token contract is INCREASED BY grams (minus fee)

xiv. internalTransferFrom

1. Access
    a. Caller MUST be EQUAL to spender
2. Parameters
    a. to

i. *to* MUST be a valid address

b.

i. *tokens* MUST be POSITIVE
ii. *tokens* MUST be LESS OR EQUAL to remaining_tokens
iii. *tokens* MUST be LESS OR EQUAL to balance

3. Output
    a. *balance* is DECREASED BY *tokens*
    b. Other fields are UNCHANGED
4. Exceptions
    i. If the constraints for the parameters ARE NOT MET
    ii. exception must be raised and NO CHANGES OCCUR

xv. disapprove

1. Access
    a. Caller MUST be a Token Owner
2. Output
    a. spender EQUALS to ZERO
    b. remaining_tokens EQUALS to ZERO
3. Exceptions
    i. If the constraints for the parameters ARE NOT MET
    ii. exception must be raised and NO CHANGES OCCUR

xvi. onBounce

1. Access
    a. Called by the system only in case of failure to deliver message
2. Parameters
    a. msg
        i. NO constraints
    b. msg_body
        i. *msg_body* MUST be a PROPER BOUNCE SLICE
        ii. BOUNCE FUNCTION from *msg_body* MUST be *internalTransfer OR internalTransferFrom*
3. Output
    a. IF BOUNCE FUNCTION from *msg_body EQUALS internalTransfer* THEN balance INCREASED BY BOUNCED VALUE from *msg_body* OTHERWISE balance is UNCHANGED
    b. All other fields ARE UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

xvii. fallback

1. Access

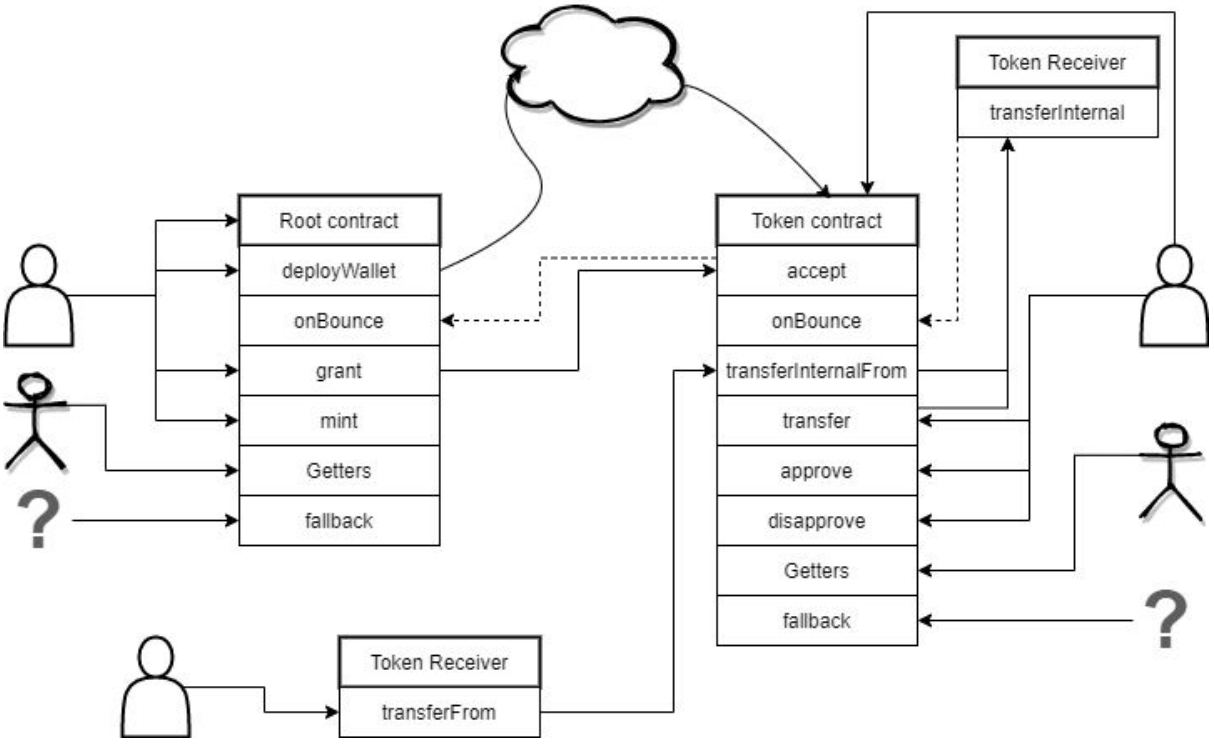
a. Called as a handler for all the internal messages other than explicitly mentioned above

2. Parameters
    a. All the parameters have NO constraints
3. Output
    a. All the fields are UNCHANGED

2. Cross-function specification
    a. All the cross-function calls are executed either as internal messages explicitly sent by origin functions or as internal messages sent by the system
    b. Any calls not mentioned in the present specification MUST not exist
    c. Recursive calls are not allowed
    d. Number of calls made by each function MUST be fixed (no calls in loops)
    e. Each message is called with *bounce* flag set to *true*
    f. In case of *bounce* the cumulative effect of the subsequent calling of the original function and *onBounce* is that all fields are UNCHANGED
    g. The full list of calls is provided below:
        i. *deployWallet*
            1. Message - *deploy* (system)
            2. Recipient - *pubkey*
            3. Parameters (name - value):
                a. *stateInfo* - CORRECT STATE INFO WITH *workchainId*, *pubKey*, *internal_owner*, name, symbol, decimals, wallet_code
            4. *value* - *grams*
        ii. *grant*
            1. Message - *accept*
            2. Recipient - *dest*
            3. Parameters
                a. *tokens* - *tokens*
            4. *value* - *grams*
        iii. *transfer*
            1. Message - *internalTransfer*
            2. Recipient - dest
            3. Parameters
                a. *tokens* - *tokens*
            4. *value* - *grams*
        iv. transferFrom
            1. Message - *internalTransferFrom*
            2. Recipient - *from*
            3. Parameters
                a. *tokens* - *tokens*
            4. *value* - *grams*
        v. *internalTransferFrom*
            1. Message - *internalTransfer*
            2. Recipient - *to*
            3. Parameters
                a. *tokens* - *tokens*

4. *value* - everything that left from incoming transfer (*SEND_REST_GAS_FROM_INCOMING*)

h. The diagram below illustrates all the internal and external messages in the system



3. Gas-related requirements
    a. All the functions getters and constructors must invoke *ACCEPT* TVM primitive immediately after correctness checking. Failure to do this may be acceptable in some cases but will require special investigation for each case
    b. Each function must spend less than *10 000* of gas

10. 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.

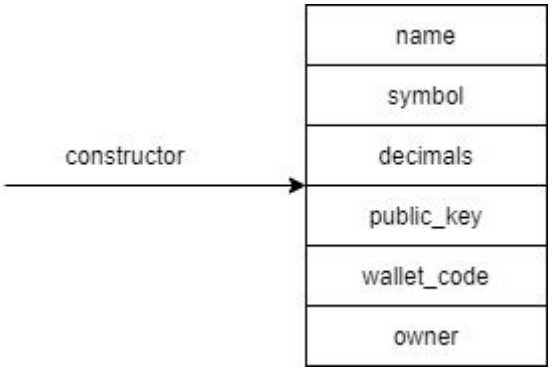
For each hypersurface the conventional way of finite state machine representation is used where squares illustrate the different state, arrows - possible transitions between states and titles for these arrows - conditions for transitions. It is worth noting that conditions have heterogeneous form and 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.

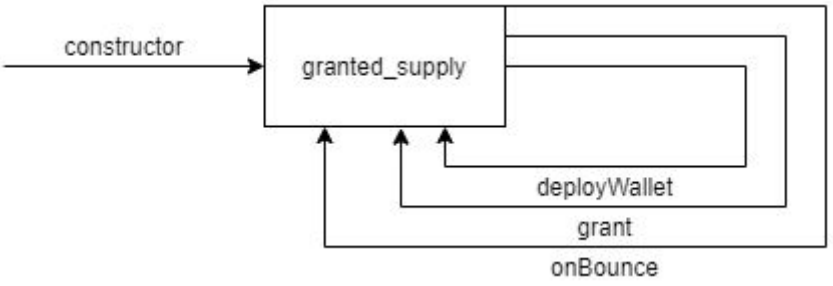
Below all the hypersurfaces are listed as well as their state-condition diagrams:

Root constant projection:



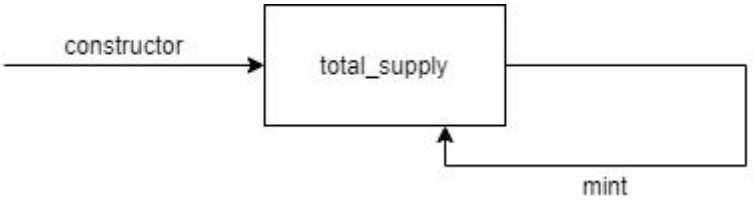
This projection represents the immutable attributes of the Root contract and corresponding diagram illustrates that these attributes can never be altered.

Root granted tokens projection:



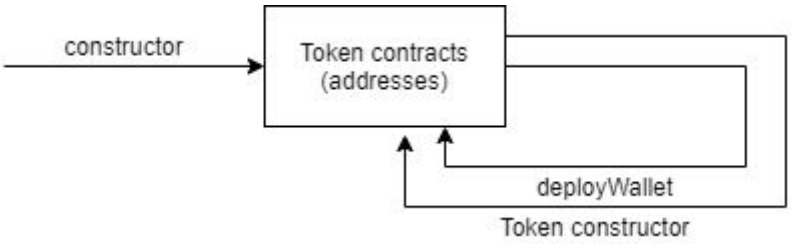
granted_supply attribute may be increased during deploying a new contract, granting tokens and decreased back in case of bouncing.

Root total supply projection:



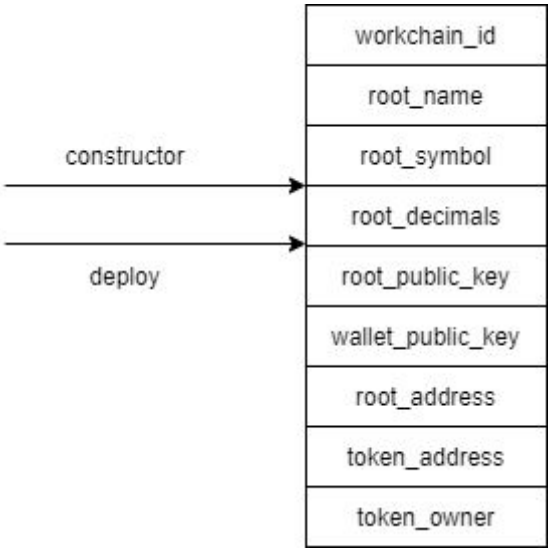
total_supply attribute may be altered exclusively when *mint* is invoked.

Root token contracts projection:



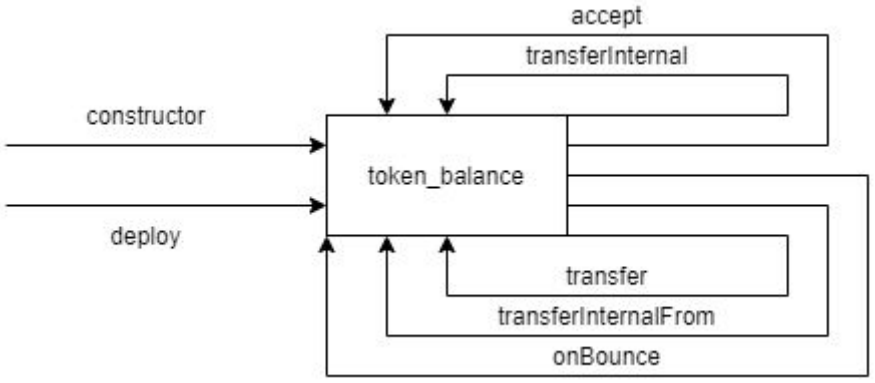
This projection represents a list of Token contracts. It's important to mention that this projection tracks the list of contracts themselves without considering their internal state (roughly speaking the list of addresses). One more thing worth noting is that this projection as well as any other uses virtual entities that can be binded to the real variables in the implementation or not. Thus the existence of such a projection doesn't suppose the existence of the corresponding collection in the implementation.

Token constants projection:



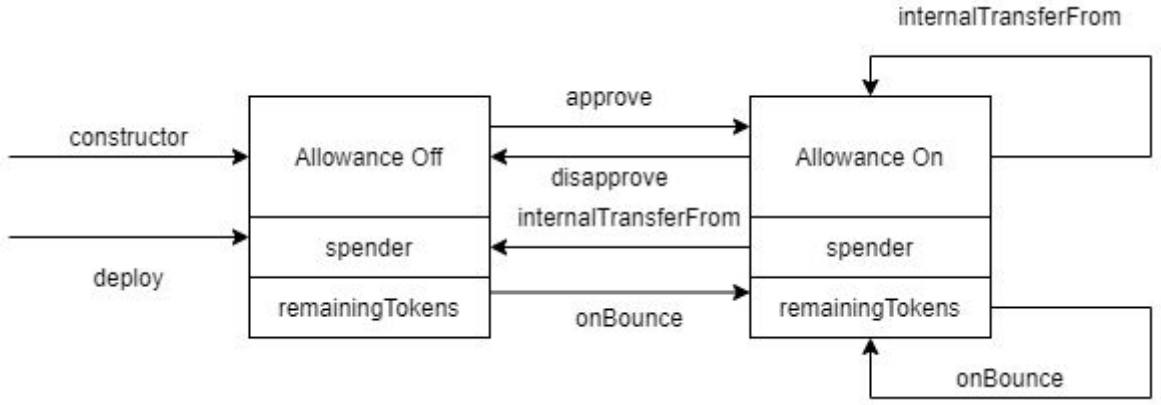
This diagram illustrates that all the immutable attributes of the Token contract can never be altered.

Token balance projection:



The token balance may be increased during *accept* or *transferInternal* and decreased at *transfer*, *transferInternalFrom* and *onBounce*.

Token allowance projection:



This projection is the only one that has two states: “Allowance Off” and “Allowance On”. Note that the state may be changed not only at direct *approve* and *disapprove* calls but also when all the remaining tokens are used and, finally, when usage of all the remaining tokens was bounced.

11. 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:

1. Root constants
    a. Construct Root contract
    b. Run Getters sequencently to check if the constants are correct
2. Bounce on deploy
    a. Construct Root contract
    b. Deploy contract to be bounced (with very low *grams*)
    c. Check *granted_tokens* increase
    d. Check *onBounce* call
    e. Check *granted_tokens* decrease to the original state
3. Bounce on grant
    a. Construct Root contract
    b. Deploy contract (with expected amount of *grams*)
    c. Check *granted_tokens* increase
    d. Grant tokens to be bounced (with very low *grams*)
    e. Check *granted_tokens* increase
    f. Check *onBounce* call
    g. Check *granted_tokens* decrease to the original state (as after point “c”)
4. Mint
    a. Construct Root contract
    b. Mint tokens
    c. Check total_supply increase
5. Token contract deployment
    a. Construct Root contract
    b. Deploy contract
    c. Construct Token contract
    d. Check by Getters that the both contracts are available
6. Deployed token contract constants
    a. Construct Root contract
    b. Deploy Token contract
    c. Run Getters sequencently to check if the constants are correct
7. Constructed token contract constants
    a. Construct Root contract
    b. Construct Token contract
    c. Run Getters sequencently to check if the constants are correct
8. Deployed bounced transfer
    a. Construct Root contract
    b. Deploy Token contract
    c. Deploy second Token contract
    d. Check *balance* of the first Token contract
    e. Grant some tokens to the first Token contract
    f. Check if *balance* of the first Token contract increased
    g. Transfer some tokens to the second contract to be bounced (with very low *grams*)

    h. Check if *balance* of the first Token contract decreased
    i. Ensure *onBounce* called
    j. Check if *balance* of the first Token contract increased (equal to the balance after execution “e” point)
9. Constructed bounced transfer
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract
    d. Check *balance* of the first Token contract
    e. Grant some tokens to the first Token contract
    f. Check if *balance* of the first Token contract increased
    g. Transfer some tokens to the second contract to be bounced (with very low *grams*)
    h. Check if *balance* of the first Token contract decreased
    i. Ensure *onBounce* called
    j. Check if *balance* of the first Token contract increased (equal to the balance after execution “e” point)
10. Deployed transfer
    a. Construct Root contract
    b. Deploy Token contract
    c. Deploy second Token contract
    d. Check *balance* of the first Token contract
    e. Grant some tokens to the first Token contract
    f. Check if *balance* of the first Token contract increased
    g. Transfer some tokens to the second contract
    h. Check if *balance* of the first Token contract decreased
    i. Check if balance of the second Token contract increased
11. Constructed transfer
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract
    d. Check *balance* of the first Token contract
    e. Grant some tokens to the first Token contract
    f. Check if *balance* of the first Token contract increased
    g. Transfer some tokens to the second contract
    h. Check if *balance* of the first Token contract decreased
    i. Check if balance of the second Token contract increased
12. Deployed bounced transfer from
    a. Construct Root contract
    b. Deploy Token contract
    c. Deploy second Token contract
    d. Deploy third Token contract
    e. Approve correct amount of token to be used by third Token contract from the first Token contract
    f. Check *balance* of the first Token contract
    g. Send “transfer from” request by the third Token contract to the first Token contract in favor of the second Token contract to be bounced at the internalTransfer call (by sending *grams* enough to cover internalTransferFrom

call fee but not exceeding amount needed to pay fee for the internalTransfer call then)
    h. Check that after *transferFrom* call no balances were changed
    i. Check that internalTransferFrom called by the first Token contract
    j. Check that after internalTransferFrom balance of the first Token contract decreased
    k. Check *onBounce* received
    l. Check that after onBounce balance of the first Token contract increased back and became equal to one after point “b”
13. Constructed bounced transfer from
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract
    d. Construct third Token contract
    e. Grant some tokens to the first contract
    f. Approve correct amount of token to be used by third Token contract from the first Token contract
    g. Check *balance* of the first Token contract
    h. Send “transfer from” request by the third Token contract to the first Token contract in favor of the second Token contract to be bounced at the internalTransfer call (by sending *grams* enough to cover internalTransferFrom call fee but not exceeding amount needed to pay fee for the internalTransfer call then)
    i. Check that after *transferFrom* call no balances were changed
    j. Check that internalTransferFrom called by the first Token contract
    k. Check that after internalTransferFrom balance of the first Token contract decreased
    l. Check *onBounce* received
    m. Check that after onBounce balance of the first Token contract increased back and became equal to one after point “e”
14. Deployed transfer from
    a. Construct Root contract
    b. Deploy Token contract
    c. Deploy second Token contract
    d. Deploy third Token contract
    e. Approve correct amount of token to be used by third Token contract from the first Token contract
    f. Check *balance* of the first Token contract
    g. Send “transfer from” request by the third Token contract to the first Token contract in favor of the second Token contract
    h. Check that after *transferFrom* call no balances were changed
    i. Check that internalTransferFrom called by the first Token contract
    j. Check that after internalTransferFrom balance of the first Token contract decreased
    k. Check *internalTransfer* called for the second Token contract
    l. Check that after internalTransfer balance of the second Token contract increased
15. Constructed transfer from

a. Construct Root contract
b. Construct Token contract
c. Construct second Token contract
d. Construct third Token contract
e. Grant some tokens to the first Token contract
f. Approve correct amount of token to be used by third Token contract from the first Token contract
g. Check *balance* of the first Token contract
h. Send “transfer from” request by the third Token contract to the first Token contract in favor of the second Token contract
i. Check that after *transferFrom* call no balances were changed
j. Check that internalTransferFrom called by the first Token contract
k. Check that after internalTransferFrom balance of the first Token contract decreased
l. Check *internalTransfer* called for the second Token contract
m. Check that after internalTransfer balance of the second Token contract increased

16. Deployed disapprove
    a. Construct Root contract
    b. Deploy Token contract
    c. Deploy second Token contract
    d. Approve correct amount of token to be used by the second Token contract from the first Token contract
    e. Check *allowanceInfo* for the first Token contract
    f. Disapprove
    g. Check *allowanceInfo* for the first Token contract
17. Constructed disapprove
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract
    d. Approve correct amount of token to be used by the second Token contract from the first Token contract
    e. Check *allowanceInfo* for the first Token contract
    f. Disapprove
    g. Check *allowanceInfo* for the first Token contract
18. Deployed allowance
    a. Construct Root contract
    b. Deploy Token contract
    c. Deploy second Token contract
    d. Approve correct amount of token to be used by the second Token contract from the first Token contract
    e. Check *allowanceInfo* for the first Token contract
    f. Invoke *transferFrom* for the second Token contract from the first Token contract in favor of the second Token contract to be bounced at the *internalTransfer* call (by sending *grams* enough to cover *internalTransferFrom* call fee but not exceeding amount needed to pay fee for the *internalTransfer* call then). The amount requested should be **less** than *remaining_tokens*.
    g. Check if *internalTransferFrom* called

    h. Check *allowanceInfo* for the first Token contract
    i. Check if *onBounce* called
    j. Check *allowanceInfo* for the first Token contract
    k. Invoke *transferFrom* for the second Token contract from the first Token contract in favor of the second Token contract to be bounced at the *internalTransfer* call (by sending *grams* enough to cover *internalTransferFrom* call fee but not exceeding amount needed to pay fee for the *internalTransfer* call then). The amount requested should **be equal** to *remaining_tokens*.
    l. Check if *internalTransferFrom* called
    m. Check *allowanceInfo* for the first Token contract
    n. Check if *onBounce* called
    o. Check *allowanceInfo* for the first Token contract
19. Constructed allowance
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract
    d. Grant some tokens to the first Token contract
    e. Approve correct amount of token to be used by the second Token contract from the first Token contract
    f. Check *allowanceInfo* for the first Token contract
    g. Invoke *transferFrom* for the second Token contract from the first Token contract in favor of the second Token contract to be bounced at the *internalTransfer* call (by sending *grams* enough to cover *internalTransferFrom* call fee but not exceeding amount needed to pay fee for the *internalTransfer* call then). The amount requested should be **less** than *remaining_tokens*.
    h. Check if *internalTransferFrom* called
    i. Check *allowanceInfo* for the first Token contract
    j. Check if *onBounce* called
    k. Check *allowanceInfo* for the first Token contract
    l. Invoke *transferFrom* for the second Token contract from the first Token contract in favor of the second Token contract to be bounced at the *internalTransfer* call (by sending *grams* enough to cover *internalTransferFrom* call fee but not exceeding amount needed to pay fee for the *internalTransfer* call then). The amount requested should **be equal** to *remaining_tokens*.
    m. Check if *internalTransferFrom* called
    n. Check *allowanceInfo* for the first Token contract
    o. Check if *onBounce* called
    p. Check *allowanceInfo* for the first Token contract

12. 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**)
- Regular bugs **(from minor to critical)**

Also should be noted that the list above covers exclusively threats internal to the *TIP-3* ecosystem itself while such external threats as private key stealing, blockchain or TVM malfunction.

13. Conclusion

The present document clearly describes all the basic scenarios to be verified, explicitly states all the potential attacks and provides the validated methodology to formally (mathematically) prove that scenarios will work as intended and attacks are not possible (or the bugs will be found and issued).

Upon completion of all the stages (not just Stage 1 described in the present document) the contract being inspected may be considered as fully reliable and safe for practical usage.

14. Company Information

Pruvendo team has been actively involved into the formal verification based on *Coq* for the last six years. During this time a number of formal verification projects have been completed, mostly in the finance and banking industry.

The team is a pioneer in mathematical justification of the proof-of-stake consensus[21], implemented the prototype of the blockchain of the formally verified code, many-years active participant of different blockchain communities.

For the last year the team has concentrated on the *TON* project and successfully proved a *Multisig* contract introducing the whole bunch of new technologies and know-hows.

Currently the team obtains a unique set of tools that lets it to quickly formally verify any kind of *TON* smart contract.

[21] https://consensusresearch.org/

Appendix A. Specification for Non-Fungible contract

1. General notes

The main difference between Non-Fungible and Fungible versions of the TIP-3 contract is if the tokens are distinguishable from each other or not. For the former they are - each token has its unique 128-bit number and each transaction is just a moving or one particular token that can not be splitted apart or combined with another one.

For the present Appendix we don't repeat any discussions that are common both for Fungible and Non-Fungible tokens, only differences are provided.

2. State

Element		Description
`name`		Unique name of the root contract
`symbol`		Unique symbol (such as BTC) of the root contract
`decimals`		
`public_key`		
`wallet_code`		Code of the Token contract as cell
`total_supply`		The overall number of tokens in the system
`granted_supply`		The overall number of tokens granted to the Token contracts
`root_token_ids`		The list of token ids owned by root
`owner`		The address who created the Root contract
`token_contracts`		List of Token contracts attached to this Root contract
	`workchain_id`	Workchain id for the Token contract
	`root_name`	Name of the Root contract
	`root_symbol`	Symbol of the Root contract
	`root_decimals`	Decimals of the Root contract
	`root_public_key`	Public key of the Root contract

	`wallet_public_key`		Public key of the Token contract
	`root_address`		Address of the Root contract
	`wallet_token_ids`		List of token ids
	`token_address`		Address of the token contract
	`token_owner`		Address of the token owner
	`allowance_info`		Structure the keeps allowance information
		`spender`	Address allowed to spend tokens
		`allowed_token_id`	Id of token allowed to spend
`world`			All the entities outside the given Root contract and the underlying Token contracts
	`world_names`		List of the root contract names other than given one
	`world_symbols`		List of the root contract symbols other than given one
	`world_public_keys`		List of the all public keys used in the blockchain other than public keys of the given Root contract and the underlying Token contracts

3. Specification in the formalized natural language

1. Function-level specification
    a. Root contract functions
        i. constructor
            1. Access
                a. NO access restrictions
            2. Parameters
                a. name
                    i. NO constraints
                b. symbol
                    i. NO constraints
                c. decimals
                    i. *decimals* MUST be NON NEGATIVE
                    ii. *decimals* MUST be LESS OR EQUAL 255
                d. root_public_key
                    i. *root_public_key* MUST be POSITIVE
                e. wallet_code
                    i. wallet_code MUST be a PROPER CODE CELL
            3. Output
                a. name EQUALS to *name*

b. symbol EQUALS to *symbol*
c. decimals EQUAL to *decimals*
d. public_key EQUALS to *root_public_key*
e. wallet_code EQUALS to *wallet_code*
f. total_supply EQUALS to ZERO
g. token_contracts MUST be EMPTY
h. root_token_ids MUST BE EMPTY
i. world is UNCHANGED
j. granted_tokens EQUAL to ZERO
k. owner MUST be the Caller

4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

ii. deployWallet

1. Access
    a. Caller MUST be an owner
2. Parameters
    a. workchain_id
        i. *workchain_id* MUST be NON NEGATIVE
    b. pub_key
        i. *pub_key* MUST be POSITIVE
        ii. pub_key MUST be UNIQUE IN token_contracts IN TERMS OF wallet_public_key AND workchain_id
    c. tokenId
        i. *tokenId* MUST be EXIST in root_token_ids
    d. grams
        i. grams MUST be POSITIVE
        ii. grams MUST be LESS OR EQUAL than the Root contract balance
3. Output
    a. name is UNCHANGED
    b. symbol is UNCHANGED
    c. decimals are UNCHANGED
    d. root_public_key is UNCHANGED
    e. wallet_code is UNCHANGED
    f. total_supply is UNCHANGED
    g. token_contracts MUST be BEING ADDED WITH **new_contract** where **new_contract** has the following attributes:
        i. workchain_id EQUALS to *workchain_id*
        ii. root_name EQUALS to name
        iii. root_symbol EQUALS to symbol
        iv. root_decimals EQUAL to decimals
        v. root_public_key EQUALS to public_key
        vi. wallet_public_key EQUALS to *pub_key*
        vii. root_address EQUALS to “My Address”
        viii. balance EQUALS to *tokens*

ix. token_address is the address of the **new_contract** as described in the section above
x. spender EQUALS to ZERO
xi. remaining_tokens EQUAL to ZERO
xii. token_owner EQUALS to internal_owner

h. world is UNCHANGED
i. granted_tokens INCREASED BY ONE
j. token_contract_ids are BEING REMOVED BY *tokenId*
k. owner is UNCHANGED

4. Exceptions
   a. If the constraints for the parameters ARE NOT MET
   b. exception must be raised and NO CHANGES OCCUR
5. Return value
   a. Address of the **new_contract**
6. Balances
   a. *grams* MUST be LESS OR EQUAL than contract balance
   b. IN CASE OF SUCCESS the contract balance is deducted by *grams* AND the balance of new contract is increased by grams MINUS fee OTHERWISE balances are UNCHANGED (but fee)

iii. grant

1. Access
   a. Caller MUST be an owner
2. Parameters
   a. address
      i. MUST EXIST in token_contracts IN TERMS OF token_address (**token_contract**)
   b. tokenId
      i. *tokenId* MUST EXIST in root_token_ids
   c. grams
      i. grams MUST be POSITIVE
      ii. grams MUST be LESS OR EQUAL than the Root contract balance
3. Output
   a. name is UNCHANGED
   b. symbol is UNCHANGED
   c. decimals are UNCHANGED
   d. root_public_key is UNCHANGED
   e. wallet_code is UNCHANGED
   f. total_supply is UNCHANGED
   g. token_contracts are UNCHANGED
   h. world is UNCHANGED
   i. granted_tokens INCREASED BY ONE
   j. owner is UNCHANGED
   k. root_token_ids HAVE *tokenId* REMOVED
4. Exceptions

a. If the constraints for the parameters ARE NOT MET
b. exception must be raised and NO CHANGES OCCUR

5. Balances
    a. *grams* MUST be LESS OR EQUAL than balance
    b. IN CASE OF SUCCESS the balance is deducted by *grams* AND the balance of new contract is increased by grams MINUS fee OTHERWISE balances are UNCHANGED (but fee) UNCHANGED

iv. mint

1. Access
    a. Caller MUST be an owner
2. Parameters
    a. tokenID
        i. *tokenID* MUST be EQUAL to token_supply PLUS ONE
3. Output
    a. total_supply is increased by ONE
    b. root_token_ids IS ADDED BY *tokenId*
    c. All the other state variables are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

v. getName

1. Access
    a. NO access restrictions
2. Return value
    a. name

vi. getSymbols

1. Access
    a. NO access restrictions
2. Return value
    a. symbol

vii. getDecimals

1. Access
    a. NO access restrictions
2. Return value
    a. decimals

viii. getRootKey

1. Access
    a. NO access restrictions
2. Return value
    a. public_key

ix. getTotalSupply

1. Access
    a. NO access restrictions
2. Return value
    a. total supply

x. getTotalGranted

1. Access
    a. NO access restrictions
2. Return value
    a. total granted

xi. getWalletCode

1. Access
    a. NO access restrictions
2. Return value
    a. wallet_code

xii. getLastMintedToken

1. Access
    a. NO access restrictions
2. Return value
    a. total_supply

xiii. getWalletAddress

1. Access
    a. NO access restrictions
2. Parameters
    a. workchain_id
        i. *workchain_id* MUST be NON NEGATIVE
    b. pub_key
        i. *pub_key* MUST EXIST IN token_contracts IN TERMS OF workchain_id AND wallet_public_key
3. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR
4. Return value
    a. wallet_public_key OF EXISTING token_contracts IN TERMS OF workchain_id AND wallet_public_key

xiv. onBounce

1. Access
    a. Called by the system only in case of failure to deliver message
2. Parameters
    a. msg
        i. NO constraints
    b. msg_body
        i. *msg_body* MUST be a PROPER BOUNCE SLICE
        ii. BOUNCED FUNCTION from *msg_body* MUST be *accept*
        iii. BOUNCED VALUE from *msg_body* MUST BE LESS OR EQUAL to total_supply
        iv. ZERO MUST BE LESS OR EQUAL THAN BOUNCED VALUE from msg_body
        v. BOUNCED VALUE from *msg_body* MUST BE UNIQUE IN root_token_ids

3. Output
    a. total_granted is DECREASED BY ONE
    b. root_token_ids IS ADDED BY BOUNCED VALUE from *msg_body*
    c. All other fields ARE UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

xv. fallback

1. Access
    a. Called as a handler for all the internal messages other than explicitly mentioned above
2. Parameters
    a. All the parameters have NO constraints
3. Output
    a. All the fields are UNCHANGED

c. Token contract functions (**NOTE!!! For this section all the values are restricted to the Token contract hypersurface unless another explicitly stated**)

i. constructor

1. Access
    a. NO access restrictions
2. Parameters
    a. name
        i. *name* EQUALS to name
    b. symbol
        i. *symbol* EQUALS to symbol
    c. decimals
        i. *decimals* EQUAL to decimals
    d. root_public_key
        i. *root_public_key* EQUALS to public_key
    e. wallet_public_key
        i. *wallet_public_key* is POSITIVE
    f. root_address
        i. *root_address* EQUALS to root_address
    g. code
        i. *code* MUST be a PROPER CODE CELL
3. Output
    a. workchain_id EQUALS to id of the current workchain
    b. root_name EQUALS to *name*
    c. root_symbol EQUALS to *symbol*
    d. root_decimals EQUALS to *decimals*
    e. root_public_key EQUALS to *root_public_key*
    f. wallet_public_key EQUALS to *wallet_public_key*
    g. root_address EQUALS to *root_address*
    h. wallet_token_ids IS EMPTY
    i. token_address EQUALS to “My Address”

j. spender EQUALS to ZERO
k. remaining_tokens EQUAL to ZERO
l. token_owner EQUALS to Caller

4. Exceptions
   a. If the constraints for the parameters ARE NOT MET
   b. exception must be raised and NO CHANGES OCCUR

ii. transfer

1. Access
   a. Caller MUST be a token_owner
2. Parameters
   a. dest
      i. *dest* MUST be a valid address
   b. tokenId
      i. *tokenId* MUST EXIST IN wallet_token_ids
   c. grams
      i. grams MUST be POSITIVE
      ii. grams MUST be LESS OR EQUAL than the Token contract balance
3. Output
   a. *tokenId* IS REMOVED from wallet_token_ids
   b. All other fields are UNCHANGED
4. Exceptions
   a. If the constraints for the parameters ARE NOT MET
   b. exception must be raised and NO CHANGES OCCUR
5. Balances
   a. *grams* MUST be LESS OR EQUAL than Token contract balance
   b. IN CASE OF SUCCESS the Sender contract balance is deducted by *grams* AND the balance of the Receiver contract balance is increased by grams MINUS fee OTHERWISE balances are UNCHANGED (but fee) UNCHANGED

iii. accept

1. Access
   a. Caller MUST be EQUAL to root_address
2. Parameters
   a. tokenId
      i. *tokenId* MUST BE UNIQUE in walet_token_ids
3. Output
   a. wallet_token_ids MUST be ADDED BY *tokenId*
   b. Other fields are UNCHANGED
4. Exceptions
   a. If the constraints for the parameters ARE NOT MET
   b. exception must be raised and NO CHANGES OCCUR

iv. internalTransfer

1. Access
   a. Caller MUST EXIST IN token_contract IN TERMS OF token_address

2. Parameters
    a. *senderKey*
        i. *senderKey* MUST EXIST IN token_contract IN TERMS OF wallet_public_key
    b. *tokenId*
        i. *tokenId* MUST BE UNIQUE in walet_token_ids
3. Output
    a. wallet_token_ids MUST be ADDED BY *tokenId*
    b. Other fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

v. *getName*
1. Access
    a. NO access restrictions
2. Return value
    a. name

vi. getSymbols
1. Access
    a. NO access restrictions
2. Return value
    a. symbol

vii. *getDecimals*
1. Access
    a. NO access restrictions
2. Return value
    a. decimals

viii. *getBalance*
1. Access
    a. NO access restrictions
2. Return value
    a. LENGTH of wallet_token_ids

ix. *getWalletKey*
1. Access
    a. NO access restrictions
2. Return value
    a. wallet_public_key

x. *getRootAddress*
1. Access
    a. NO access restrictions
2. Return value
    a. root_address

xi. *allowance*
1. Access
    a. NO access restrictions
2. Return value
    a. allowance_info

xii. *getTokenByIndex*

1. Access
    a. NO access restrictions
2. Parameters
    a. index
        i. *index* MUST BE LESS THAN LENGTH of wallet_token_ids
        ii. *index* MUST BE NON NEGATIVE
3. Return value
    a. *index*-th element of wallet_token_ids

xiii. getApproved

1. Access
    a. NO access restrictions
2. Parameters
    a. tokenId
        i. *tokenId* MUST EXIST IN wallet_token_ids
        ii. *tokenId* MUST be EQUAL to allowed_token_id
3. Return value
    a. spender
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

xiv. approve

1. Access
    a. Caller MUST be a token_owner
2. Parameters
    a. spender
        i. *spender* MUST be a valid address
    b. tokenId
        i. *tokenId* MUST EXIST IN wallet_token_ids
3. Output
    a. spender EQUALS to *spender*
    b. allowed_token_id EQUALS TO *tokenId*
    c. Other fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

xv. transferFrom

1. Access
    a. Caller MUST be a Token Owner
2. Parameters
    a. dest
        i. *dest* MUST be a valid address
    b. to
        i. *to* MUST be a valid address
    c. tokenId
        i. *tokenId* MUST be POSITIVE
    d. grams
        i. *grams* MUST be POSITIVE

3. Output
    a. All the fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR
5. Balances
    a. Balance of the Token contract is INCREASED BY grams (minus fee)

xvi. internalTransferFrom

1. Access
    a. Caller MUST be EQUAL to spender
2. Parameters
    a. to
        i. *to* MUST be a valid address
    b.
        i. *tokenId* MUST be EQUAL to allowed_token_id
        ii. *tokenId* MUST EXIST IN wallet_token_ids
3. Output
    a. *tokenId* IS REMOVED from wallet_token_ids
    b. Other fields are UNCHANGED
4. Exceptions
        i. If the constraints for the parameters ARE NOT MET
        ii. exception must be raised and NO CHANGES OCCUR

xvii. disapprove

1. Access
    a. Caller MUST be a Token Owner
2. Output
    a. spender EQUALS to ZERO
    b. allowed_token_id EQUALS to ZERO
3. Exceptions
        i. If the constraints for the parameters ARE NOT MET
        ii. exception must be raised and NO CHANGES OCCUR

xviii. onBounce

1. Access
    a. Called by the system only in case of failure to deliver message
2. Parameters
    a. msg
        i. NO constraints
    b. msg_body
        i. *msg_body* MUST be a PROPER BOUNCE SLICE
        ii. BOUNCE FUNCTION from *msg_body* MUST be *internalTransfer OR internalTransferFrom*

3. Output
    a. IF BOUNCE FUNCTION from *msg_body EQUALS internalTransfer* THEN wallet_token_ids MUST be ADDED BY BOUNCED VALUE from *msg_body* OTHERWISE balance is UNCHANGED
    b. All other fields ARE UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

xix. fallback

1. Access
    a. Called as a handler for all the internal messages other than explicitly mentioned above
2. Parameters
    a. All the parameters have NO constraints
3. Output
    a. All the fields are UNCHANGED

4. Cross-function specification
    a. All the cross-function calls are executed either as internal messages explicitly sent by origin functions or as internal messages sent by the system
    b. Any calls not mentioned in the present specification MUST not exist
    c. Recursive calls are not allowed
    d. Number of calls made by each function MUST be fixed (no calls in loops)
    e. Each message is called with *bounce* flag set to *true*
    f. In case of *bounce* the cumulative effect of the subsequent calling of the original function and *onBounce* is that all fields are UNCHANGED
    g. The full list of calls is provided below:

i. deployWallet

1. Message - *deploy* (system)
2. Recipient - *pubkey*
3. Parameters (name - value):
    a. *stateInfo* - CORRECT STATE INFO WITH *workchainId*, *pubKey*, *internal_owner*, name, symbol, decimals, wallet_code
4. *value* - *grams*

ii. grant

1. Message - *accept*
2. Recipient - *dest*
3. Parameters
    a. *tokenId* - *tokenId*
4. *value* - *grams*

iii. transfer

1. Message - *internalTransfer*
2. Recipient - dest
3. Parameters
    a. *tokenId* - *tokenId*
4. *value* - *grams*

iv. transferFrom

1. Message - *internalTransferFrom*
2. Recipient - *from*
3. Parameters
    a. *tokenId* - *tokenId*
4. *value* - *grams*

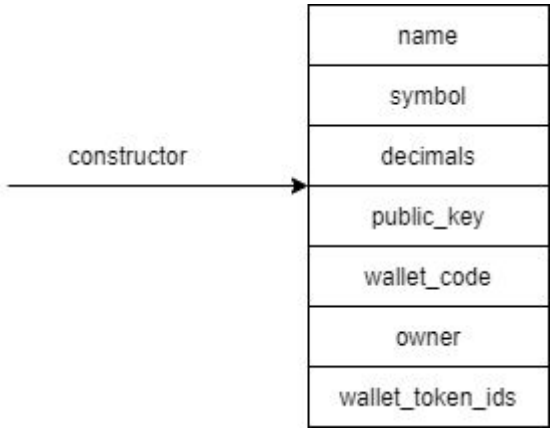
v. *internalTransferFrom*

1. Message - *internalTransfer*
2. Recipient - *to*
3. Parameters
    a. *tokenId* - *tokenId*
4. *value* - everything that left from incoming transfer (*SEND_REST_GAS_FROM_INCOMING*)

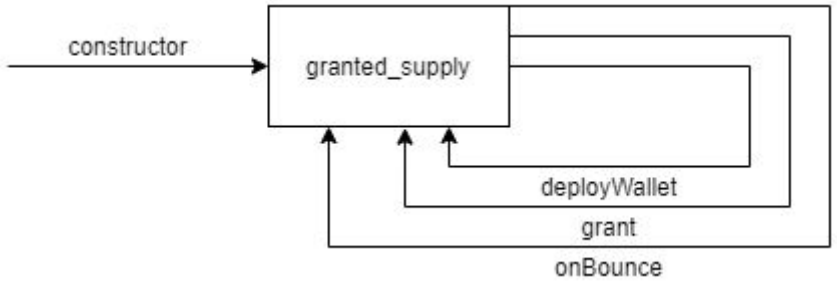
h. The diagram below illustrates all the internal and external messages in the system

4. Projections

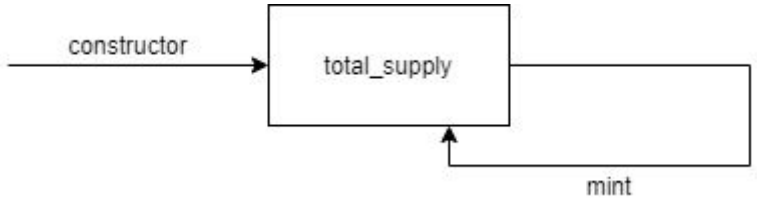
Root constant projection:



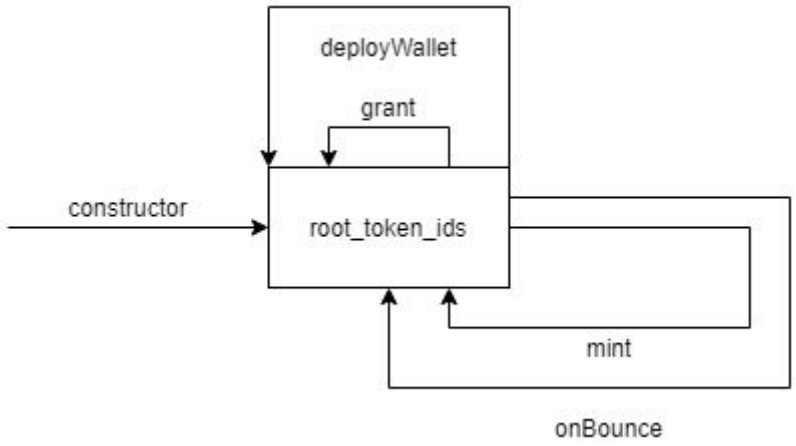
Root granted projection:



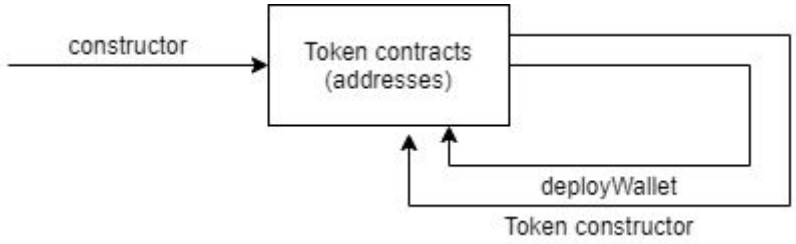
Root total projection:



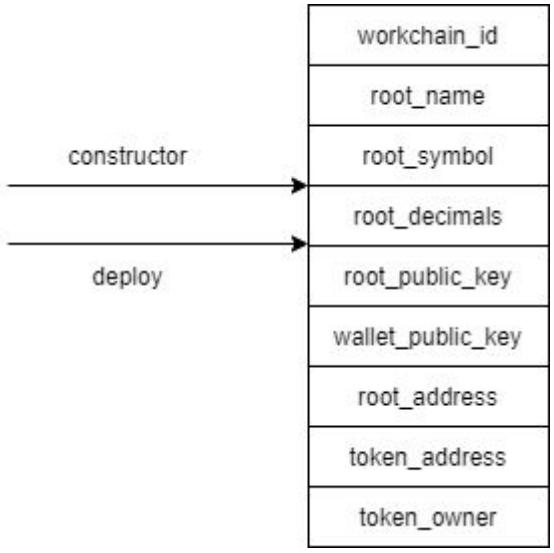
Root token ids projection:



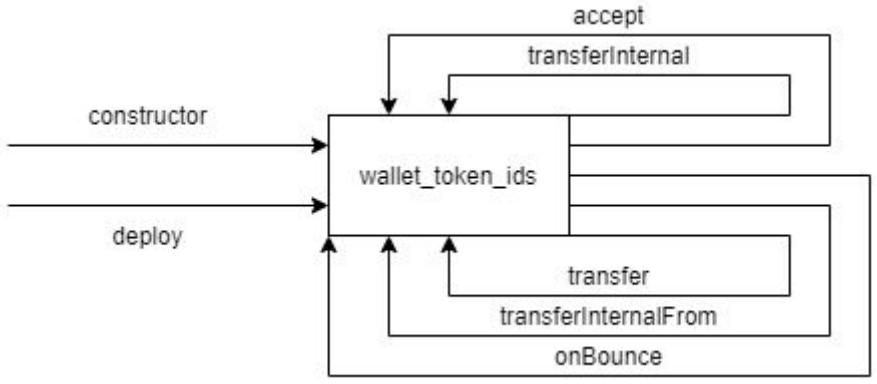
Root token contracts projection:



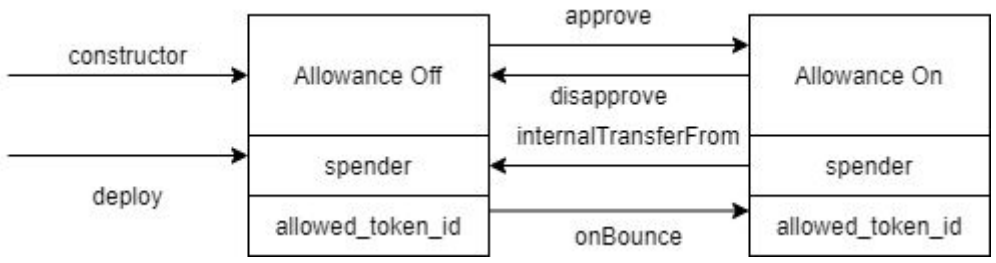
Token constants projection:



Token ids projection:



Token allowance projection:



5. Scenarios

1. Root constants
    a. Construct Root contract
    b. Run Getters sequencently to check if the constants are correct
2. Bounce on deploy
    a. Construct Root contract
    b. Mint token
    c. Deploy contract to be bounced (with very low *grams*)
    d. Check *granted_tokens* increase
    e. Check *onBounce* call
    f. Check *granted_tokens* decrease to the original state
3. Bounce on grant
    a. Construct Root contract
    b. Mint token
    c. Deploy contract (with expected amount of *grams*)
    d. Check *granted_tokens* increase
    e. Grant token to be bounced (with very low *grams*)
    f. Check *granted_tokens* increase
    g. Check *onBounce* call
    h. Check *granted_tokens* decrease to the original state (as after point “c”)
4. Mint
    a. Construct Root contract
    b. Mint token
    c. Check *total_supply* increase
5. Mint, deploy, grant and onBounce
    a. Construct Root contract
    b. Mint token
    c. Check *root_token_ids*
    d. Deploy Token contract
    e. Check *root_token_ids*
    f. Mint token
    g. Check *root_token_ids*
    h. Grant token providing very low amount of *grams*
    i. Check *root_token_ids*
    j. Check *onBounce* received
    k. Check *root_token_ids*
6. Token contract deployment
    a. Construct Root contract

    b. Mint token
    c. Deploy contract
    d. Construct Token contract
    e. Check by Getters that the both contracts are available
7. Deployed token contract constants
    a. Construct Root contract
    b. Deploy Token contract
    c. Run Getters sequencently to check if the constants are correct
8. Constructed token contract constants
    a. Construct Root contract
    b. Construct Token contract
    c. Run Getters sequencently to check if the constants are correct
9. Deployed bounced transfer
    a. Construct Root contract
    b. Mint two tokens
    c. Deploy Token contract
    d. Deploy second Token contract
    e. Check *wallet_token_ids* of the first Token contract
    f. Grant token to the first Token contract
    g. Check *wallet_token_ids* of the first Token contract
    h. Transfer token to the second contract to be bounced (with very low *grams*)
    i. Check *wallet_token_ids* of the first Token contract
    j. Ensure *onBounce* called
    k. Check *wallet_token_ids* of the first Token contract
10. Constructed bounced transfer
    a. Construct Root contract
    b. Mint token
    c. Construct Token contract
    d. Construct second Token contract
    e. Check *wallet_token_ids* of the first Token contract
    f. Grant token to the first Token contract
    g. Check *wallet_token_ids* of the first Token contract
    h. Transfer some tokens to the second contract to be bounced (with very low *grams*)
    i. Check *wallet_token_ids* of the first Token contract
    j. Ensure *onBounce* called
    k. Check *wallet_token_ids* of the first Token contract
11. Deployed transfer
    a. Construct Root contract
    b. Mint two tokens
    c. Deploy Token contract
    d. Deploy second Token contract
    e. Check *wallet_token_ids* of the first Token contract
    f. Grant token to the first Token contract
    g. Check *wallet_token_ids* of the first Token contract
    h. Transfer token to the second contract
    i. Check *wallet_token_ids* of the first Token contract
    j. Check *wallet_token_ids* of the second Token contract

12. Constructed transfer
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract
    d. Check *wallet_token_ids* of the first Token contract
    e. Mint token
    f. Grant token to the first Token contract
    g. Check *wallet_token_ids* of the first Token contract
    h. Transfer token to the second contract
    i. Check *wallet_token_ids* of the first Token contract
    j. Check *wallet_token_ids* of the second Token contract
13. Deployed bounced transfer from
    a. Construct Root contract
    b. Mint three tokens
    c. Deploy Token contract
    d. Deploy second Token contract
    e. Deploy third Token contract
    f. Approve token owned by the first contract to be used by third Token contract from the first Token contract
    g. Check *wallet_token_ids* of the first Token contract
    h. Send “transfer from” request by the third Token contract to the first Token contract in favor of the second Token contract to be bounced at the internalTransfer call (by sending *grams* enough to cover internalTransferFrom call fee but not exceeding amount needed to pay fee for the internalTransfer call then)
    i. Check that after *transferFrom* call no *wallet_token_ids* were changed
    j. Check that internalTransferFrom called by the first Token contract
    k. Check *wallet_token_ids* of the first Token contract after internalTransferFrom
    l. Check *onBounce* received
    m. Check *wallet_token_ids* of the first Token contract after onBounce
14. Constructed bounced transfer from
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract
    d. Construct third Token contract
    e. MInt token
    f. Grant token to the first contract
    g. Approve granted token to be used by third Token contract from the first Token contract
    h. Check *wallet_token_ids* of the first Token contract
    i. Send “transfer from” request by the third Token contract to the first Token contract in favor of the second Token contract to be bounced at the internalTransfer call (by sending *grams* enough to cover internalTransferFrom call fee but not exceeding amount needed to pay fee for the internalTransfer call then)
    j. Check that after *transferFrom* call no *wallet_token_ids* were changed
    k. Check that internalTransferFrom called by the first Token contract
    l. Check *wallet_token_ids* of the first Token contract

m. Check *onBounce* received
n. Check *wallet_token_ids* of the first Token contract

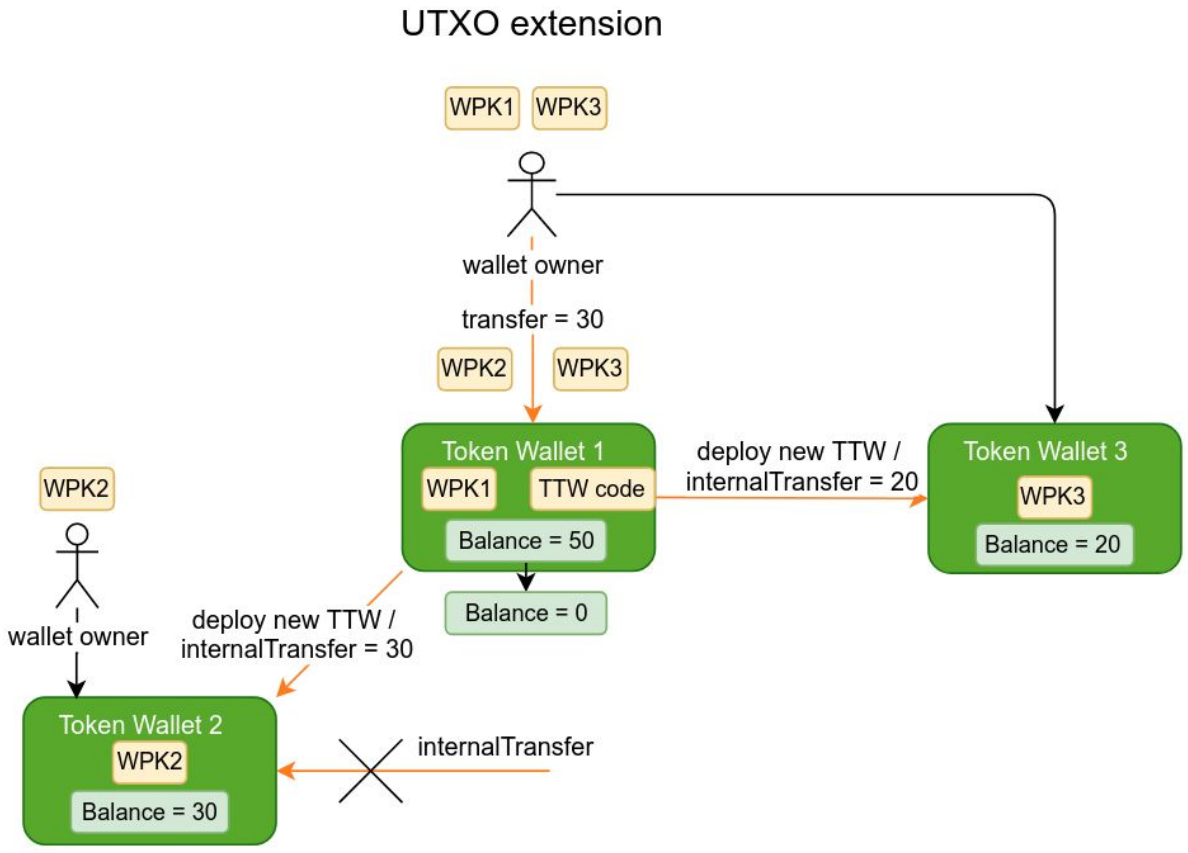
15. Deployed transfer from
    a. Construct Root contract
    b. Mint three tokens
    c. Deploy Token contract
    d. Deploy second Token contract
    e. Deploy third Token contract
    f. Approve token owned by the first contract to be used by third Token contract from the first Token contract
    g. Check *wallet_token_ids* of the first Token contract
    h. Send “transfer from” request by the third Token contract to the first Token contract in favor of the second Token contract
    i. Check that after *transferFrom* call no *wallet_token_ids* were changed
    j. Check that internalTransferFrom called by the first Token contract
    k. Check *wallet_token_ids* of the first Token contract after internalTransferFrom call
    l. Check *internalTransfer* called for the second Token contract
    m. Check *wallet_token_ids* of the second Token contract
16. Constructed transfer from
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract
    d. Construct third Token contract
    e. Approve token owned by the first contract to be used by third Token contract from the first Token contract
    f. Check *wallet_token_ids* of the first Token contract
    g. Send “transfer from” request by the third Token contract to the first Token contract in favor of the second Token contract
    h. Check that after *transferFrom* call no wallet_token_ids were changed
    i. Check *internalTransferFrom* was called
    j. Check *wallet_token_ids* of the first Token contract after internalTransferFrom
    k. Check *internalTransfer* called for the second Token contract
    l. Check *wallet_token_ids* of the second Token contract
17. Deployed disapprove
    a. Construct Root contract
    b. Mint two tokens
    c. Deploy Token contract
    d. Deploy second Token contract
    e. Approve token to be used by the second Token contract from the first Token contract
    f. Check *allowanceInfo* for the first Token contract
    g. Disapprove
    h. Check *allowanceInfo* for the first Token contract
18. Constructed disapprove
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract

    d. Mint token
    e. Approve token to be used by the second Token contract from the first Token contract
    f. Check *allowanceInfo* for the first Token contract
    g. Disapprove
    h. Check *allowanceInfo* for the first Token contract
19. Deployed allowance
    a. Construct Root contract
    b. Mint two tokens
    c. Deploy Token contract
    d. Deploy second Token contract
    e. Approve token owned by the first Token contract to be used by the second Token contract from the first Token contract
    f. Check *allowanceInfo* for the first Token contract
    g. Invoke *transferFrom* for the second Token contract from the first Token contract in favor of the second Token contract to be bounced at the *internalTransfer* call (by sending *grams* enough to cover *internalTransferFrom* call fee but not exceeding amount needed to pay fee for the *internalTransfer* call then).
    h. Check if *internalTransferFrom* called
    i. Check *allowanceInfo* for the first Token contract
    j. Check if *onBounce* called
    k. Check *allowanceInfo* for the first Token contract
20. Constructed allowance
    a. Construct Root contract
    b. Construct Token contract
    c. Construct second Token contract
    d. Mint token
    e. Grant token to the first Token contract
    f. Approve token to be used by the second Token contract from the first Token contract
    g. Check *allowanceInfo* for the first Token contract
    h. Invoke *transferFrom* for the second Token contract from the first Token contract in favor of the second Token contract to be bounced at the *internalTransfer* call (by sending *grams* enough to cover *internalTransferFrom* call fee but not exceeding amount needed to pay fee for the *internalTransfer* call then).
    i. Check if *internalTransferFrom* called
    j. Check *allowanceInfo* for the first Token contract
    k. Check if *onBounce* called
    l. Check *allowanceInfo* for the first Token contract

Appendix B. Specification for UTXO contract

1. General notes

UTXO contract is a specific variation of Fungible contract where Token contract is somehow “immutable”. It can not be topped up (so no equivalent of *grant* is available) as well as it doesn’t support allowance. In case of transfer the contract is splitted into two - one belongs to the “new” owner with the transferred amount of tokens and the second one belongs to the “original” owner with the remaining balance. The original contract is not destroyed but “retired” with zero balance. The picture below illustrates the business logic of the UTXO contract.



2. State

Element	Description
`name`	Unique name of the root contract
`symbol`	Unique symbol (such as BTC) of the root contract
`decimals`	

public_key		
wallet_code		Code of the Token contract as cell
total_supply		The overall number of tokens in the system
granted_supply		The overall number of tokens granted to the Token contracts
owner		The address who created the Root contract
token_contracts		List of Token contracts attached to this Root contract
	workchain_id	Workchain id for the Token contract
	root_name	Name of the Root contract
	root_symbol	Symbol of the Root contract
	root_decimals	Decimals of the Root contract
	root_public_key	Public key of the Root contract
	wallet_public_key	Public key of the Token contract
	root_address	Address of the Root contract
	balance	Balance of the token contract
	token_address	Address of the token contract
	token_owner	Address of the token owner
	utxo_received	Indicates if UTXO received for the contract
world		All the entities outside the given Root contract and the underlying Token contracts
	world_names	List of the root contract names other than given one
	world_symbols	List of the root contract symbols other than given one
	world_public_keys	List of the all public keys used in the blockchain other than public keys of the given Root contract and the underlying Token contracts

3. Specification in the formalized natural language

1. Function-level specification
    a. Root contract functions

i. *constructor*
    1. Access
        a. NO access restrictions
    2. Parameters
        a. *name*
            i. NO constraints
        b. *symbol*
            i. NO constraints
        c. *decimals*
            i. *decimals* MUST be NON NEGATIVE
            ii. *decimals* MUST be LESS OR EQUAL 255
        d. *root_public_key*
            i. *root_public_key* MUST be POSITIVE
        e. *wallet_code*
            i. wallet_code MUST be a PROPER CODE CELL
        f. *total_supply*
            i. *total_supply* MUST be POSITIVE
    3. Output
        a. name EQUALS to *name*
        b. symbol EQUALS to *symbol*
        c. decimals EQUAL to *decimals*
        d. public_key EQUALS to *root_public_key*
        e. wallet_code EQUALS to *wallet_code*
        f. total_supply EQUALS to total_supply
        g. token_contracts MUST be EMPTY
        h. world is UNCHANGED
        i. granted_tokens EQUAL to ZERO
        j. owner MUST be the Caller
    4. Exceptions
        a. If the constraints for the parameters ARE NOT MET
        b. exception must be raised and NO CHANGES OCCUR

ii. *deployWallet*
    1. Access
        a. Caller MUST be an owner
    2. Parameters
        a. *workchain_id*
            i. *workchain_id* MUST be NON NEGATIVE
        b. *pub_key*
            i. *pub_key* MUST be POSITIVE
            ii. pub_key MUST be UNIQUE IN token_contracts IN TERMS OF wallet_public_key AND workchain_id
        c. *tokens*
            i. *tokens* MUST be POSITIVE
            ii. *tokens* MUST be LESS OR EQUAL than total_supply MINUS granted_supply
        d. grams
            i. grams MUST be POSITIVE

ii. grams MUST be LESS OR EQUAL than the Root contract balance

3. Output
    a. name is UNCHANGED
    b. symbol is UNCHANGED
    c. decimals are UNCHANGED
    d. root_public_key is UNCHANGED
    e. wallet_code is UNCHANGED
    f. total_supply is UNCHANGED
    g. token_contracts MUST be BEING ADDED WITH **new_contract** where **new_contract** has the following attributes:
        i. workchain_id EQUALS to *workchain_id*
        ii. root_name EQUALS to name
        iii. root_symbol EQUALS to symbol
        iv. root_decimals EQUAL to decimals
        v. root_public_key EQUALS to public_key
        vi. wallet_public_key EQUALS to *pub_key*
        vii. root_address EQUALS to “My Address”
        viii. balance EQUALS to *ZERO*
        ix. token_address is the address of the **new_contract** as described in the section above
        x. token_owner EQUALS to internal_owner
        xi. utxo_received EQUALS to FALSE
    h. world is UNCHANGED
    i. granted_tokens INCREASED BY *tokens*
    j. owner is UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR
5. Return value
    a. Address of the **new_contract**
6. Balances
    a. *grams* MUST be LESS OR EQUAL than contract balance
    b. IN CASE OF SUCCESS the contract balance is deducted by *grams* AND the balance of new contract is increased by grams MINUS fee OTHERWISE balances are UNCHANGED (but fee)

iii. mint

1. Access
    a. Caller MUST be an owner
2. Parameters
    a. tokens
        i. *tokens* MUST be POSITIVE
3. Output
    a. total_supply is increased by *tokens*

b. All the other state variables are UNCHANGED

4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

iv. getName
1. Access
    a. NO access restrictions
2. Return value
    a. name

v. getSymbols
1. Access
    a. NO access restrictions
2. Return value
    a. symbol

vi. getDecimals
1. Access
    a. NO access restrictions
2. Return value
    a. decimals

vii. getRootKey
1. Access
    a. NO access restrictions
2. Return value
    a. public_key

viii. getTotalSupply
1. Access
    a. NO access restrictions
2. Return value
    a. total supply

ix. getTotalGranted
1. Access
    a. NO access restrictions
2. Return value
    a. total granted

x. getWalletCode
1. Access
    a. NO access restrictions
2. Return value
    a. wallet_code

xi. getWalletAddress
1. Access
    a. NO access restrictions
2. Parameters
    a. workchain_id
        i. *workchain_id* MUST be NON NEGATIVE
    b. pub_key

i. *pub_key* MUST EXIST IN token_contracts IN TERMS OF workchain_id AND wallet_public_key

3. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR
4. Return value
    a. wallet_public_key OF EXISTING token_contracts IN TERMS OF workchain_id AND wallet_public_key

xii. fallback

1. Access
    a. Called as a handler for all the internal messages other than explicitly mentioned above
2. Parameters
    a. All the parameters have NO constraints
3. Output
    a. All the fields are UNCHANGED

b. Token contract functions (**NOTE!!! For this section all the values are restricted to the Token contract hypersurface unless another explicitly stated**)

i. constructor - **this method is considered as useless so it's moved out the specification**

ii. transferUTXO

1. Access
    a. Caller MUST be a token_owner
2. Parameters
    a. workchain_dest
        i. *workchain_dest* MUST be NON NEGATIVE
    b. pubkey_dest
        i. *pubkey_dest* MUST be POSITIVE
    c. workchain_rest
        i. *workchain_rest* MUST be NON NEGATIVE
    d. pubkey_rest
        i. *pubkey_rest* MUST be POSITIVE
    e. tokens
        i. *tokens* MUST be POSITIVE
        ii. *tokens* MUST be LESS OR EQUAL than balance
    f. grams
        i. grams MUST be POSITIVE
        ii. grams MUST be LESS OR EQUAL than the Token contract balance
3. Output
    a. balance EQUALS TO ZERO
    b. All the other fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET

b. exception must be raised and NO CHANGES OCCUR

5. Balances
    a. *grams* MUST be LESS OR EQUAL than Token contract balance
    b. IN CASE OF SUCCESS the Sender contract balance is deducted by *grams* AND the balance of the both Receivers contract balance is spread by grams MINUS fee OTHERWISE balances are UNCHANGED (but fee) UNCHANGED

iii. accept

1. Access
    a. Caller MUST be EQUAL to root_address
2. Parameters
    a. tokens
        i. *tokens* MUST be POSITIVE
3. Output
    a. balance MUST be INCREASED BY *tokens*
    b. utxo_received EQUALS to TRUE
    c. Other fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

iv. internalTransfer

1. Access
    a. Caller MUST EXIST IN token_contract IN TERMS OF token_address
2. Parameters
    a. senderKey
        i. senderKey MUST EXIST IN token_contract IN TERMS OF wallet_public_key
    b. tokens
        i. *tokens* MUST be POSITIVE
3. Output
    a. balance MUST be EQUAL to *tokens*
    b. utxo_received EQUALS to TRUE
    c. Other fields are UNCHANGED
4. Exceptions
    a. If the constraints for the parameters ARE NOT MET
    b. exception must be raised and NO CHANGES OCCUR

v. getName

1. Access
    a. NO access restrictions
2. Return value
    a. name

vi. getSymbols

1. Access bb
    a. NO access restrictions
2. Return value

a. symbol

vii. *getDecimals*
1. Access
   a. NO access restrictions
2. Return value
   a. decimals

viii. *getBalance*
1. Access
   a. NO access restrictions
2. Return value
   a. balance

ix. *getWalletKey*
1. Access
   a. NO access restrictions
2. Return value
   a. wallet_public_key

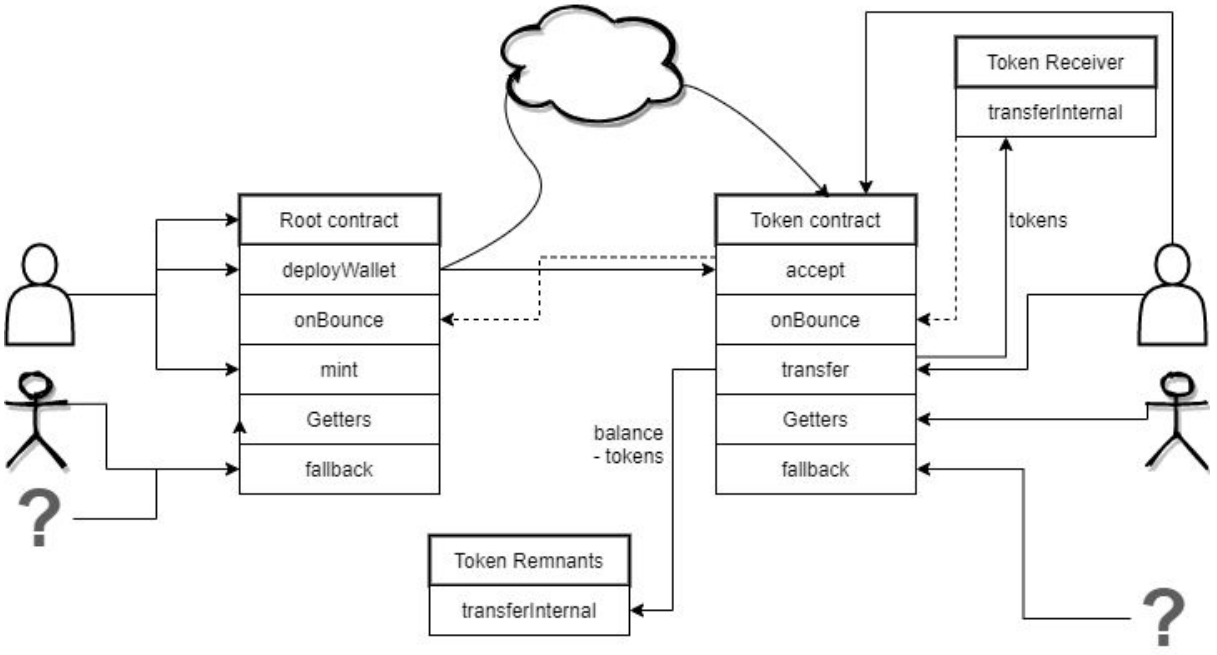
x. *getRootAddress*
1. Access
   a. NO access restrictions
2. Return value
   a. root_address

xi. *onBounce*
1. Access
   a. Called by the system only in case of failure to deliver message
2. Parameters
   a. msg
      i. NO constraints
   b. msg_body
      i. *msg_body* MUST be a PROPER BOUNCE SLICE
      ii. BOUNCE FUNCTION from *msg_body* MUST be *internalTransfer OR internalTransferFrom*
3. Output
   a. balance INCREASED BY BOUNCED VALUE from *msg_body*
   b. All other fields ARE UNCHANGED
4. Exceptions
   a. If the constraints for the parameters ARE NOT MET
   b. exception must be raised and NO CHANGES OCCUR

xii. *fallback*
1. Access
   a. Called as a handler for all the internal messages other than explicitly mentioned above
2. Parameters
   a. All the parameters have NO constraints
3. Output

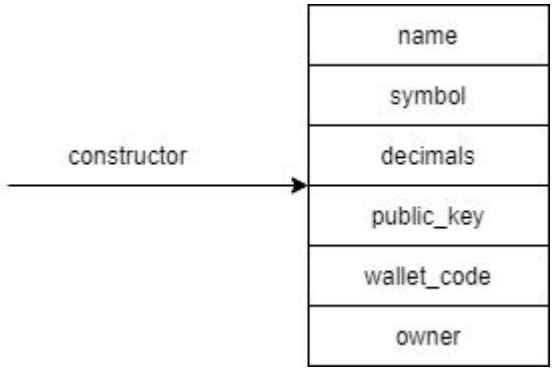
a. All the fields are UNCHANGED

2. Cross-function specification
    a. All the cross-function calls are executed either as internal messages explicitly sent by origin functions or as internal messages sent by the system
    b. Any calls not mentioned in the present specification MUST not exist
    c. Recursive calls are not allowed
    d. Number of calls made by each function MUST be fixed (no calls in loops)
    e. Each message is called with *bounce* flag set to *true*
    f. In case of *bounce* the cumulative effect of the subsequent calling of the original function and *onBounce* is that all fields are UNCHANGED
    g. The full list of calls is provided below:
        i. deployWallet
            1. Message - *deploy* (system)
            2. Recipient - *pubkey*
            3. Parameters (name - value):
                a. *stateInfo* - CORRECT STATE INFO WITH *workchainId*, *pubKey*, *internal_owner*, name, symbol, decimals, wallet_code
            4. *value* - *grams*
        ii. transfer
            1. Message - *internalTransfer*
            2. Recipients - *dest* & *rest*
            3. Parameters
                a. dest
                    i. *tokens* - *tokens*
                b. rest
                    i. tokens - *balance* MINUS *tokens*
            4. *value* - *grams*
    h. The diagram below illustrates all the internal and external messages in the system

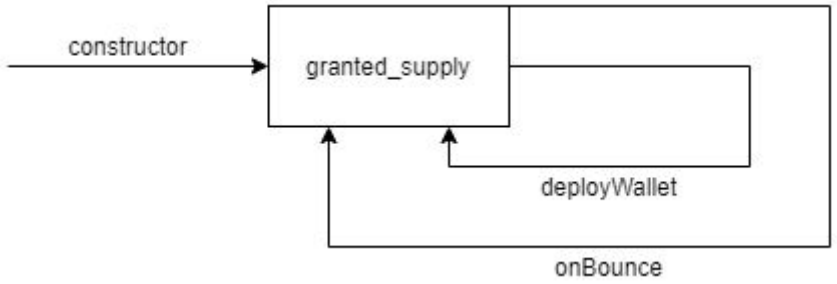


4. Projections

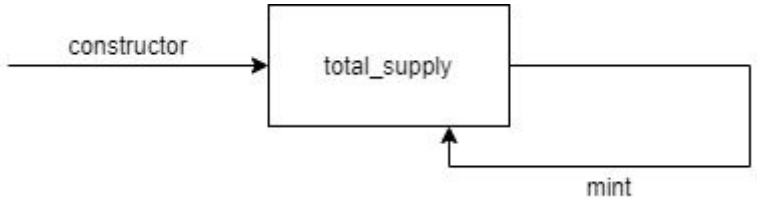
Root constants projection:



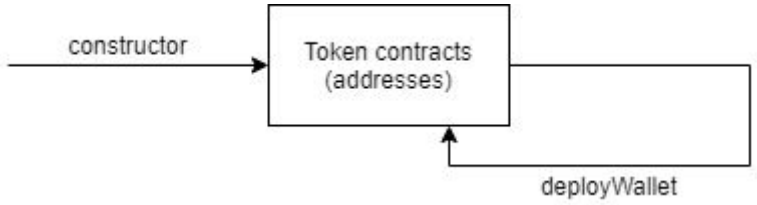
Root granted projection:



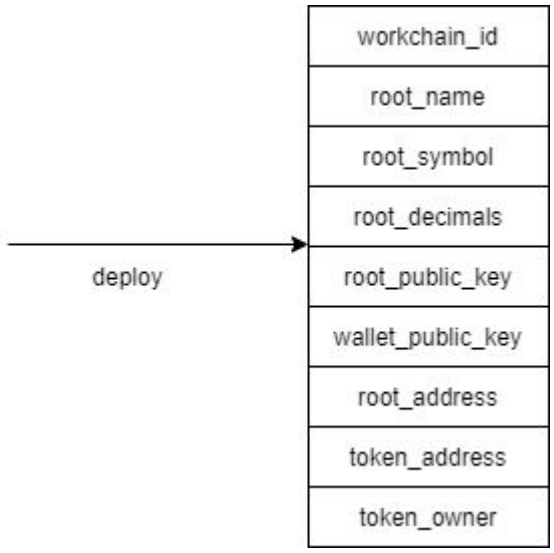
Root total projection:



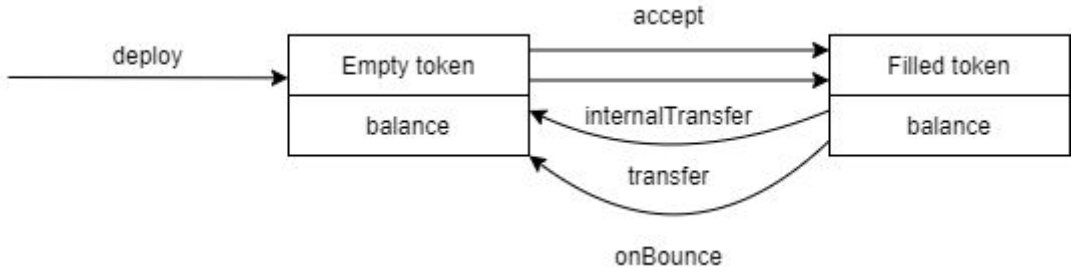
Root tokens projection:



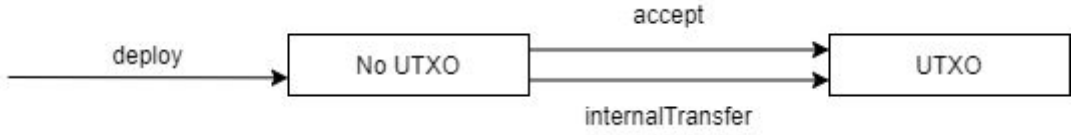
Token constants projection:



Token balance projection:



Token UTXO projection:



5. Scenarios

1. Root constants
    a. Construct Root contract
    b. Run Getters sequencently to check if the constants are correct
2. Bounce on deploy
    a. Construct Root contract
    b. Deploy contract to be bounced (with very low *grams*)
    c. Check *granted_tokens* increase
    d. Check *onBounce* call
    e. Check *granted_tokens* decrease to the original state
3. Mint
    a. Construct Root contract
    b. Mint tokens
    c. Check *total_supply* increase
4. Token contract deployment
    a. Construct Root contract
    b. Deploy contract

    c. Check by Getters that the contract is available
5. Deployed token contract constants
    a. Construct Root contract
    b. Deploy Token contract
    c. Run Getters sequencently to check if the constants are correct
6. Deployed bounced transfer
    a. Construct Root contract
    b. Deploy Token contract
    c. Check *balance* of the Token contract
    d. Transfer some tokens to the another address to be bounced (with very low *grams*)
    e. Check if *balance* of the first Token contract is zero
    f. Ensure *onBounce* called
    g. Check if *balance* of the first Token contract increased (equal to the balance after execution “c” point)
7. Deployed transfer
    a. Construct Root contract
    b. Deploy Token contract
    c. Check *balance* of the Token contract
    d. Transfer some tokens to the some address
    e. Check if *balance* of the Token contract is ZERO
    f. Check if balance of the Token contract identified by *dest* increased
    g. Check if balance of the Token contract identified by *rest* increased