Formal verification for Hipo project (Wallet contract)

Prepared by Pruvendo at 06/04/24

Executive summary

Pruvendo performed a formal verification (as an advanced form of an audit that dramatically decreases chance for important bugs, in a light form known as quickchicks) for Wallet smart contracts of Hipo project (https://github.com/HipoFinance/contract/blob/main/contracts/wallet.fc, commit eb839658c954c485060bd8d9cf838bbe93f18f75, made on 03/22/24),

A number of **suspicious issues** were found, provided in the present report were discussed with developers and found to be addressed by the upper contracts.

While no real issues were found, the present report is intended to describe an extremely thorough and reliable approach that decreases the probability of bugs virtually to zero.

Description of formal verification approach

Smart contracts, which handle large amounts of money and are part of the open Internet, are frequently targeted by hackers with advanced attack techniques (such as the Lazarus group, which is responsible for up to 35% of the overall stolen funds in web3). For example, the Pruvendo team discovered a project with a hidden vulnerability. Despite having only $3 in liquidity and no prior announcement, the project was targeted by automated hacking systems.

Smart contracts differ from traditional software by being grounded in decentralization principles and not providing total system control. This often leads to the inability to resolve the consequences of incorrect program operation, resulting in unpleasant effects, up to the freezing of funds.

Industry experts estimate that the web3 sector suffers substantial financial losses, amounting to billions of dollars annually, as a result of errors in smart contracts. Pruvendo is aware of over 170 cases with a total damage exceeding 6 billion dollars.

Considering the circumstances, traditional quality assurance methods (QA), including architectural methods, best programming practices, code review, multi-level testing, post-production monitoring, etc., remain necessary but not sufficient to ensure the required code quality.

Traditionally, the smart contract industry relies on audits to tackle this issue. However, the effectiveness of this approach is not consistently satisfactory. For instance, in the midst of the intense work phase on the FreeTon project (now known as Everscale), there were instances where exceptionally skilled audit teams couldn't identify a single shared bug in one

smart contract (**list of found vulnerabilities of every team was completely different to each other**), highlighting the limitations of manual audits.

The **fundamental solution to the problem is formal verification**, namely, a mathematical (in the form of a set of theorems and lemmas) method of proving program correctness (or rather, its compliance with specifications).

This approach was developed about half a century ago but remained too complex and expensive for practical use, having a niche solely for high-budget projects (such as the Paris metro, the Hercules military transport aircraft, etc.).

However, in recent years, a number of projects have been carried out aiming to adapt the aforementioned concept for the mass market.

Pruvendo is one of the few companies in the formal verification of software market that has managed to adapt the most comprehensive and complete approach for the broader industry application - **deductive verification**, which is completely strict in the mathematical sense of the term (roughly speaking, implementation correctness is proved as a theorem).

The result of formal verification is formal proof. Proof is a program that can be independently verified using a process known as Typechecking. This process is performed in a distinct system - Proof Assistant. This is the reason why the results of formal verification do not require additional validation. Proof shows 100% correspondence of implementation to specification (in the mathematical sense of the term). Thus, any incorrect behavior of the program can be caused only by an incorrect specification. To reduce the risk of incorrect specification, Pruvendo has developed its own technology for specification design (it is based on a multistage approach that allows it to deep down gradually thus observing the whole project from multiple points of view).

The outcome of the formal verification is a comprehensive **range of services that results in**:

- **Proof** of minimal chance of errors (bug-free guarantees are impossible because of risks of mistakes in specifications)
- Or a **detailed list of encountered mistakes** along with their degree of significance

Thus, the formal verification is effectively splitted into two big tasks:

- Formal specification
- Formal verification itself

An approach for both tasks will be considered below.

Formal specification approach

Introduction

When formal verification is being discussed it's important to understand that it's an approach to mathematically prove not the *correctness* of the particular software program (*correctness* is just a common sense term that means nothing in a formal world (roughly speaking, it's the same as *do the job right*, but what *right* means?)), but rather it's about proving the equivalence (or, stricter speaking, isomorphism) between the **implementation** being verified and some entity called **formal specification**.

There are many formal specification approaches such as TLA+ , Z notation etc. sometimes based on rather advanced technologies such as λ-calculus, however, in the opinion of *Pruvendo*, all of them have the following critical drawbacks that prevents them from using in the industrial formal verification:

- The customer (usually, software architect) need to dive too deep in the world of the new concepts that prevents him from the adequate assessing the provided specification, thus increasing risk of errors from the specifier
- There is a high risk of missing some important specification statements, thus increasing the possibility that some important bugs remain unfound

The drawbacks mentioned above significantly decrease an ability of the formal verification to become a magic bullet for the mission-critical software.

Pruvendo suggests its own originally developed approach that intents to:

- Provide step-to-step move from the business-level specification to the formal one to let the customer to stop at the stage of comfort and understanding
- Help to the specifier not to forget anything, thus getting the full specification rather than one that misses the critical statements

Mathematically, the suggested approach has the strong mathematical base on Category theory, toposes and monads, however, its understanding is not required for reading the present document.

The present document is intended to help the reader to understand the whole concepts as well as to get the particular specification up to the comfort stage.

Top-level specification

Business-level specification

The specification creation starts from the business-level specification. It's just a text document that describes:

- Purpose of the project

- Key concepts
- The detailed description of the project behavior

This document is supposed to be understandable by any customer and must be approved by him to ensure everything is caught by the specifier correctly.

Stage 1. Scenarios

The key idea of the presented paradigm is the scenario-based approach. While the scenario is a logically bound useful interaction with the software (or, in some cases, elements of such an interaction), the whole specification is considered as a set of scenarios (rather independent to each other).

The task of choosing if a particular logically bound construction is a scenario or not is a sole decision of the specifier (can be roughly considered with a decision to put some software code into a separate file or not).

An example of a scenario system is provided in the Appendix A, that should help to understand what scenario means.

High-level specification

High-level specification is intended to provide step-to-step formal specification without being tightly bound to the implementation. Thus, it's important to mention that some entities defined in the high-level specification **don't directly correspond to the implementation entities**. Such a gap is to be resolved at the low level of the specification.

Technological notes

Currently, the high-level specification technologically is based on draw.io/diagrams.net with some additional Kotlin-based proprietary tools to perform some transformations of the diagrams as well as some extra auxiliary activity.

In the near future *Pruvendo* plans to move to own toolchain.

Stage 2. Output

The first stage of each scenario is to identify the outputs. Indeed, nobody from software development follows Porthos, a character from the novel of Dumas, who fought just for fighting. The reason (output) of each scenario is the only justification for its existence. The following graphical elements present here:

Rectangle	Description of the *outcome* in a natural language. Different rectangles stay for different outcomes (outcome results?)
Parentheses	Combine the different *rectangles* into the same *outcome result*

Out	The terminal element of the *Output*. Its meaning is under discussion, may be removed in future
Arrows	The connecting lines between the core part of the *Output* and *Out*. May be removed in future

Stage 3. Input

When the *Output* is defined, it's time to define *Input*, so the set of *Actors* that initiate the scenario. The following types of *Actors* may exist (titles correspond to graphical images in a specification):

- *Human* - it's an external actor that commonly acts from direct instructions of human being or a software that pretends to be made from protein
    - Humans can be different (say, *Owner* is a different *Human* than just a *User*)
- *Cloud* - it's an external actor that presents an oracle - off-chain software module that somehow manages the workflow for smart contracts
- *Triangle*(autostart) - it's an on-chain actor (the higher scenario) that triggers the action. This kind of *Actors* can indicate the upper scenario (or scenarios) that may trigger it

Each *Actor* can trigger one or more actions defined by the corresponding arrows linked with the action name.

Stage 4. Body

Stage 4a. States I

To be sure that nothing is forgotten the concept of states has been introduced. In the most simple case there are just two states: *Nothing* (aka 'before action') and *Created* (aka 'after action'). In case of more complex scenarios the list of states can be more thorough. As an example, consider a software bug lifetime, say, in Jira: it can contain dozens of different states (say, *Created*, *Accepted*, *Evaluated*, …).

In the same way, a number of different states can be defined here. However, it's important to state that it's just a helping stage that may be skipped.

Stage 4b. Main body

When the *Output* (what is intended to reach) and *Input* (what initiated the scenario) it's time to define the stuff in the middle, called *Body*. Normally, it (the Body) consists of six types of graphical elements:

- Set of rectangles (*decision matrix*):
    - Set of exception conditions in a natural form
    - *Otherwise*, as a positive path

- Framed rectangles - we call them *masks*, empty as this stage, will be explained below
- Named rectangles - at this stage just a name of the particular transformation of data, without details. We call them *transformation elements*
- Bold rectangles - *subscenarios*, to be defined later in the same way as regular scenarios
- Crossed circles - we call them *mask cancellation*
- Connecting arrows

It's important to mention that at this step we define the scenario specification in the form of its skeleton, without any details.

Stage 5. Details

While the previous stages define the skeleton of the scenario, the present stage finalizes high-level specification by setting data types, data objects, their relations and transformations.

Roughly speaking, it's a full specification not bound to the implementation.

Stage 5a. Types and objects

As in many classic programming languages, the *types* and *objects* are introduced here. Types can be primitive, sets, and custom.

The following primitive types are defined:

Type	Meaning
A	Address. It's important to mention that this type is not obliged to correspond to, say, *address* type in Solidity, but just represents some entity that is unique for any corresponding object
B	Boolean. Just a regular logical type with two objects: *true* and *false*
N	Unsigned number. It's important to mention that this type corresponds to the mathematical $\aleph \cup \{0\}$ extended natural set. All the upper boundaries typically are not subject to high-level specification
S	Scenario type. Corresponds to the set of scenarios and subscenarios
UUID	Similar to *A*, can be used when the specifier wants to distinguish entities of a different nature

Like in most programming languages, *sets* are unordered homogeneous collections of the objects of particular type. It is worth mentioning that currently no inheritance is supported, so the strict type equivalence is assumed.

Custom types are usually heterogeneous structures that unite objects of different nature and type.

There are some built-in (in terms of specification system) objects:

Object	Meaning
l	It's an object of the *custom* type *Ledger*. Basically, it's a root object of the whole blockchain state, where only objects important for the scenario being considered are identified
s	It's an *A*-typed object that sends the initial message (aka *msg.sender*)
ss	It's a *S*-typed object that invokes the scenario being considered (actual for subscenarios only)

The system of types and objects for the particular scenario is represented by a mixed graph, that is supposed to be intuitively clear for understanding, with the following assumptions:

- Types are represented by ellipses
- Objects are represented by the named connection arrays
- Single objects (not sets) are represented by thin arrays
- Sets are represented by thick arrays

Some built-in functions and shortcuts are also introduced for specific objects:

- For any *A*-object - *b* (or *balance*) (`<no parameters>`) - balance of the objects in terms of the native currency (such as *ETH* or *TRX*)
- For any *A*-object that corresponds to non-fungible token (such as *ERC20* or *TRC20*):
    - It can be directly accessed by its name enclosed in apostrophes (such as *'USDT'*)
    - It has the following functions:
        - *b* (or *balance*) (`<owner:A>`) - token balance of the `owner`
        - *a* (or *allowance*) (`<allower:A, allowee:A>`) - allowance of the `allower` to the `allowee`

Disclaimer!!! All the types and objects being described in the present section may or may not correspond to the types and objects of the implementation. While it's recommended to somehow follow the implementation to make it more understandable and ease the creation of low-level stuff, the final decision is up to the decision of the specifier.

Stage 5b. States II

This auxiliary step helps to transform the purely descriptive Stage I step into the detailed one. Its sole purpose is to provide better understanding of the specification.

Technically the step being described is a copy of Stage I with the detailed description (in terms of relations between objects) what each particular stage means.

Stage 5c. Details

When the types and objects are defined (see Stage 5a) it's time to complete high-level specification by adding object workflow, restrictions and transformations into the scenario skeleton acquired at the Stages 2-4.

Let's start with the *Input* part:

- If the particular *Actor* is restricted (other than an arbitrary actor, as an example, *Owner*) it's accompanied with a formula that shows if the *Actor* is allowed to perform the specific action or not
- Input arguments:
    - All the input arguments are enclosed by cylinders, where, upon their first appearance, their type is directly provided in the corresponding callouts
    - If the arguments come from the upper scenarios, they initially appear in the arrow preceding the *Actor* or the whole *Input* part
    - Otherwise, the initially appear at the action arrow

The *Body* part is the most tricky one:

- All the input arguments (including newly created ones) follow the same rule as above
- *Decision matrices* are extended with formulae that provide a logical value if the particular condition is met or not
- *Masks* are extended with the list of entities that **can** be modified. The following rules are in place:
    - If any item of a structure (object if the *Custom* type) is modified, the structure itself is considered as not modified
- *Transformation elements* are extended with the list of transformation rules. Among the common sense rules (plain formulae) the following syntax is used:
    - For sets, the following notations are used:
        - *s*+=*x* - the set is rather unchanged, but one more element is added
        - *s*-=*x* - the set is rather unchanged, but one element is removed
    - For primitive types, the following notations are used:
        - *a*+=*x* - *a* is increased by *x* as a result of the transformation
        - *a*-=*x* - *a* is decreased by *x* as a result of the transformation
    - For all other cases:
        - Object without apostrophe - before transformation
        - Object with apostrophe - after transformation

For the *Output* part, normally nothing is changed, otherwise the same notations are used.

Stage 5d. Invariants

While graphical specification is considered as more friendly, understandably and full than traditional ones, the verifiers work with statements, not the lines.

So the final step is to convert the pictures into a set of mathematical expressions. Surprisingly, this activity is almost mechanical and will eventually be performed fully automatically. To understand this step one must:

- Read thoroughly the previous sections
- Understand the Hoare logic. It's simple:
    - *{A}B{C}* - must be read as:
        - If predicate *A*:
        - Then, after transformation *B*:
        - *C*
    - $\frac{A}{B} \Leftrightarrow (A \Leftrightarrow B)$
    - Basically, it just defines the state before and after some particular transformation
- To handle the case '*nothing else is changed*' the special character is introduced (*crossed up arrow*), that means that a particular predicate does not depend on the particular object or its descendants (in terms of structures)

Upon the completion of this step, high-level specification is fully defined, however, for the formal verification, a mapping of the specification entities (types, objects, statements) to the implementation ones is required, however, it's considered in a section below.

Stage 6: Low-level specification

Unlike high-level specification that is somehow agnostic about the implementation, low-level specification puts all the statements aligned with implementation entities (such as contracts, structures and variables) . Such an activity leads to generation of three following tables for each scenario (the first one can be single for a group of scenarios).

Axioms

It's a table that provides a set of statements that are accepted without proof. Typically, they are either related to:

- Language-specific behavior
- Restrictions from outer systems (such as web3 applications)
- Features of the external smart contracts not to be verified (such as, say, external staking system)

The resulting table for axioms has three columns:

- *Axiom ID* - two letters (that are supposed to describe a domain for the axiom), dot and number (such as *EI.2*)
- *Human description* - description of the axiom in English
- *Formal description* - mathematical formal description, where:
    - Hoare logic is used, as above
    - Unlike in high-level specification, implementation entities are used rather than abstract concept entities

Also, some global (inside the range of the particular scenario or the set of scenarios) first-order logic predicates can be used in the third column leaving the former ones empty.

Mapping

Moving to low-level specification it's important to map all the high-level entities to the implementation-specific ones. The corresponding table is auxiliary, but helps to understand the correspondence between high and low levels. The mapping table has the following columns:

- Name of the high-level entity
- Name of the corresponding low-level entity
- Comments

It's important to highlight that this section is intended exclusively for better understanding and not to be used explicitly by verifiers.

Low-level invariants

It's the main section of the low-level specification, intended for direct usage of verifiers and serving as an ultimate result of specifiers. Technically, it's a table with the following columns:

- *Statement ID* - mapped from the high-level specification
- *Human description* - mapped from the high-level specification
- *Formal description* - can be either:
    - empty and gray, if the corresponding high-level statement is not applicable anymore
    - transformed from high-level representation into low-level one using mapping discussed above[1]

As above:

- Hoare logic is used
- Some scenario-wide (or more local) predicates can be used in separate lines

Stage 7: Conversion to Coq

When all the statements are fully specified, the transformation of them into Coq statements is nothing more but a routine activity, such as changing of ∀ to `forall`, while Hoare logic is fully supported by Coq. Currently this activity is manual, but it's planned to be fully automated[2].

Upon completion of this stage, the goal of the formal specification is fully completed - it is:

- Understandable by any PM or team lead without need to learn something new, until the last stages, that can be fully automated

[1] Currently this process is manual, the partial or full automation is planned in foreseeable future
[2] Automation of *Stage7* is in the roadmap

- Full, as a detailed multi-step process brings a probability of missing something to a minimal value
- Correct, as the developers are able to understand it, as mentioned above

As a summary, the described process virtually eliminates the Achilles' heel of the formal verification - pool form specification.

Formal verification approach

Conversion to Ursus

Most of the smart contract languages, such as Solidity, are imperative, while the proof assistants, such as Coq, are based on declarative languages, such as OCaml. To resolve this issue a special intermediate language called *Ursus* has been developed. Technically it's a DSL over Gallina (OCaml dialect used by Coq) that it's syntactically very close to Solidity. So, just see the following example:

Solidity code	Ursus code

`function _deleteUpdateRequest(uint64 updateId, uint8 index) inline private {`
`  m_updateRequestsMask &= ~(uint32(1) << index);`
`  delete m_updateRequests[updateId];`
`}`
	`#[private, nonpayable]`
	`Ursus Definition _deleteUpdateRequest (updateId : uint64) (index : uint8):`
	`UExpression PhantomType false .`
	`{`
	`  ::// m_updateRequestsMask &= ( ~ ( (uint32(1) << {index}))) .`
	`  ::// m_updateRequests[updateId] ->delete |`
	`}`
	`return.`
	`Defined.`
	`Sync.`

It's easy to see that while the syntaxes are different, they can be mutually mapped from Solidity to Ursus and vice versa.

Currently the following translators to Ursus are implemented:

- TVM Solidity (Everscale, GOSH)
- Classic Solidity (Ethereum, Tron, other EVM-compatible chains)
- Rust (MultiversX)
- FunC (TonCoin)

Ursus also can be used as a primary language for development, with later translation to Solidity. Such an approach makes a formal verification easier, as Ursus prevents a

developer from introducing some code patterns that bring some complication for the verifications.

The detailed Ursus specification and source code of translators can be provided by request.

Evals&Execs

While imperative code is successfully turned into declarative one, the state can be wrapped into a state monad, where each method is considered as a combination of two functions that correspondingly return:

- *Eval* - return value of the method
- *Exec* - the modified state

Pruvendo has developed a tool called *Generator* that performs automatic generation of *evals* and *execs* for the most practical cases.

Verification

All the previous steps can be considered as preparation for this one. Indeed, by this stage the two main artifacts are available:

- Formal specification as a set of Coq statements
- Implementation as a set of Coq functions

So, at this point nothing prevents the verifiers from proving that the implementation corresponds to the specification that is the ultimate goal of the formal verification.

To simplify this process a number of so-called Coq *tactics* (proof steps) has been developed and currently this process is semi-automated, while still requires involvement of high-skill professionals.

By the end of this stage the set (isomorphic to each low-level specification statements) of the Coq statements (treated as lemmas or theorems) is either:

- *Proved* - that means a full correspondence of the implementation to the specification, in this case the result of the verification process is positive
- *Can not be proven* - in this case the obstacle that prevents the statements from being proven is to be identified and discussed with the development team. Finally, the obstacle is treated either as:
    - Specification bug - specification is to be changed
    - Minor note - specification is to be changed, while this note must be reflected in the final report
    - Major note (bug) - the implementation must be fixed, otherwise the positive verdict is not granted

As a result of the described process the probability of the undiscovered critical bugs becomes extremely low that allows to claim a system that successfully passed the formal verification process as **reliable**.

Quickchicks (light-weight formal verification)

While the systems of smart contracts with complicated business logic and/or implementation leave no other options, for simple systems a lighter approach with straightforward logic can be used. The idea is to use *QuickChick* randomized property-based Coq plugin that verifies the properties not by strict mathematical proving but by providing random input data checking.. It's worth mentioning that this approach, while inferior to deductive verification, is still much more powerful than traditional random testing, as predicates allow to automatically discover classes of equivalence while the classic approach fully relies on the operator's expertise.

Formal verification

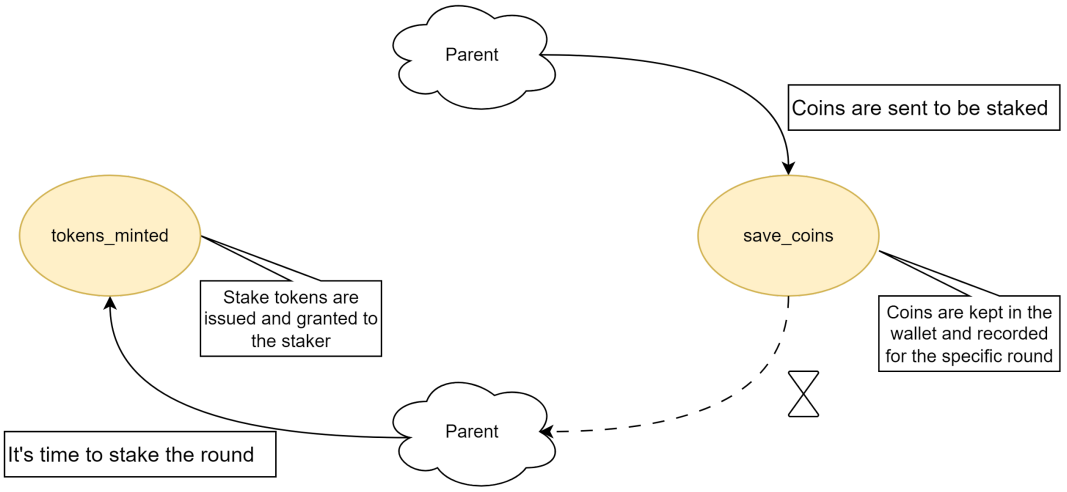
Business-level specification

Purpose

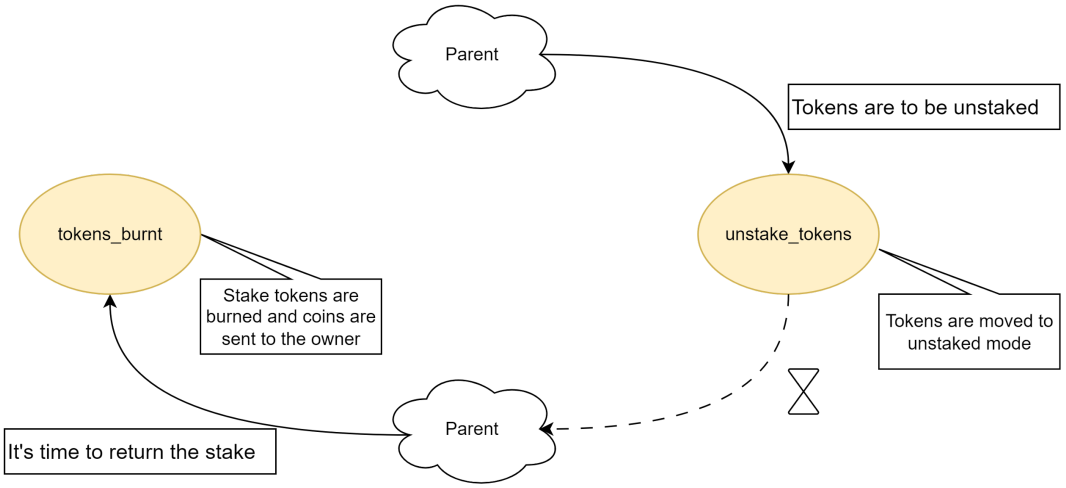
The purpose of the present document is to specify the *Wallet* contract at a business level and to use it as a basement for the development of the formal specification, that, in its turn, will be used for the formal verification of *Wallet*.

Introduction

Wallet is a standard TON jetton wallet with some additional capability - staking support that can be roughly illustrated by the following diagrams:



Dig 1. Principles of wallet staking



Dig.2 Principles of wallet unstaking

So, the main workflow looks as follows:

- Staking:
  - *Parent* (a contract that is not considered in the present document) sends some coins[3] to the *Wallet* using *save_coins* message
  - *Wallet* freezes the received coins and adds the received amount to the amount assigned for a specific round of staking[4]
  - Eventually, *Parent* claims the readiness of the round by sending tokens_minted message
  - *Wallet*:
    - Cleans a record about amount assigned for this round of staking
    - Mints the corresponding amount of tokens
- Unstaking:
  - *Parent* send a request to the Wallet using *unstake_tokens* message
  - *Wallet* Temporarily locks the required amount of tokens
  - Unless the unstaking is not rolled back, eventually the *Parent* sends *tokens_burned* message

The main workflow is described above, but a number of additional activities are also possible. All of them are claimed below and considered in details in the following section:

- Common Jetton activities (as described by Jetton standard):
  - Send tokens
- Secondary activities:
  - Minting without staking
  - Rollback unstaking
  - Unstaking all
  - Upgrade start
  - Upgrade completion

[3] Here and later *coin* stands for native Toncoin

[4] Staking tokens are not minted immediately, but rather wait for running the specified so called “round of staking”.

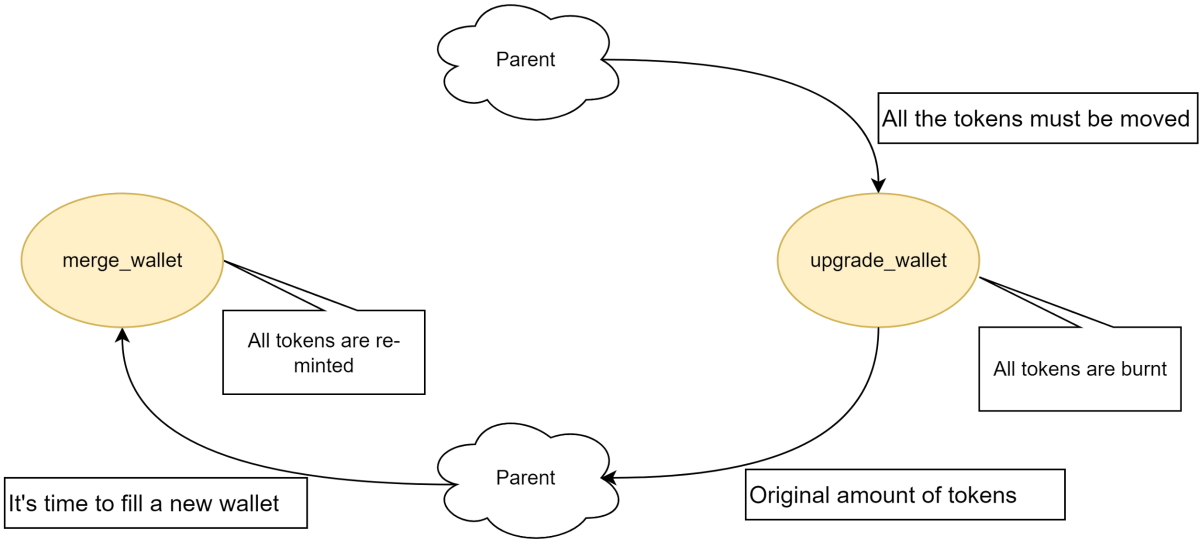
    - Withdraw extra coins
    - Token transfer initiated by *Parent*
- System activities:
    - Dispatching
    - Bounce

All these activities are discussed below in detail.

The special attention has to be paid to upgrade workflow, that works as follows:

- At first, Parent initiates this activity by sending *upgrade_wallet* message
- *Wallet* burns all the tokens and sends their original amount back to the Parent
- *Parent* eventually sends *merge_wallet* message to a **new** *Wallet*
- **new** *Wallet* mints the required amount of tokens, thus resembling an old, but upgraded one

Graphically the process can be illustrated by the following diagram:



Dig.3 Upgrade

Detailed description

Main workflow

Staking

save_coins

This subscenario:

- receives “information about amount of coins” (info) from *Parent*, as well as a staking round
- Adds the info to an amount assigned for this round, if exists
- Otherwise, simply assigns an amount assigned for this round to the info

The message can be sent by *Parent* only, amount of coins must be positive

tokens_minted

This subscenario:

- receives amount to mint from *Parent*, as well as a staking round and query id
- If an amount assigned for the round is insufficient, or does not exist, it fails
- Otherwise:
    - Subtracts an amount required from the amount assigned for the round
    - Mints the required amount of tokens
- Secondary: staking round can be not provided. In this case the required amount of tokens is simply minted
- In all the cases but failure a notification message is sent to the *Owner*[5], containing the following information:
    - query id
    - amount
    - *Owner*

The message can be sent by *Parent* only, also the required amount must be positive.

Unstaking

unstake_tokens

This subscenario:

- Receives the following data from *Parent* or *Wallet* itself:
    - query id
    - amount of tokens
    - address to return excess (extra gas)
    - mode (its meaning is out of the *Wallet* contract)
    - additional fee
- The operation fails, if and only if:
    - The sender is neither *Parent* not *Wallet itself*
    - Fee can not be paid (standard + additional)
    - Amount of unlocked tokens is not sufficient
    - Request amount is zero
    - Unexpected mode
- If the operation successes:
    - The requested amount of tokens is locked
    - Notification message *proxy_reserve_tokens* is sent to the *Parent* with the following data:
        - query id
        - amount of tokens
        - *Owner*
        - additional fee

[5] *Owner* is a smart-contract that represents an owner of the *Wallet* (often it's an avatar of an off-chain entity (human being or an oracle))

tokens_burned

This subscenario:

- Receives the following data from *Parent*:
  - query id
  - token amount to write off
  - amount of coins to be sent to the *Owner*
- The operation fails if and only if:
  - The sender is not the *Parent*
  - There is insufficient amount of coins for transfer (including fee)
  - Insufficient amount of locked tokens
- In case of success:
  - Locked amount of tokens is decreased by the specified amount
  - A message to the *Owner* with the specified amount of coins and the following data is sent:
    - query id
    - amount of tokens
    - amount of coins

Send & Receive

These operations are related to standard Jetton token transfer. For convenience, some requirements of this standard are copied here, rather than just providing a link to them.

send_tokens

This subscenario:

- Receives the following data:
  - query id
  - amount to send
  - recipient
  - address to return extra gas
  - amount of coins to transfer with
  - custom payload
- The operation fails if and only if:
  - The sender is not the *Owner*
  - There is no enough coins to pay fee (and payload of coins) that are calculated as:
    - Standard fee, if there is no coins to transfer with
    - Double fee added by coins to transfer
  - There is an insufficient amount of tokens to transfer
  - Recipient is the sender
- In case of success:
  - Amount of tokens is decreased by the transfer amount
  - *receive_tokens* message is sent with the following data:
    - query id
    - amount of tokens
    - *Owner*

        - address to return extra gas
        - amount of coins to transfer with
        - custom payload

receive_tokens

This subscenario:

- Receives the following data:
    - query id
    - amount of tokens
    - *Owner*
    - address to return extra gas
    - amount of coins to transfer with
    - custom payload
- The operation fails if and only if:
    - The sender is not the *Owner*
- In case of success:
    - Amount of tokens is increased by the transfer amount
    - In case of non-zero amount of coins to transfer with, the notification is sent to *Owner* (according to the Jetton standard) with the following data:
        - query id
        - amount
        - sender
        - custom payload

Secondary actions

Minting without staking

This activity is fully covered at tokens_minted

Rollback unstaking

This subscenario:

- Receives the following data:
    - query id
    - amount of tokens to be rolled back
- The operation fails if and only if:
    - The sender is not *Parent*
    - Amount of locked tokens exceeds the required amount
- In case of success:
    - Amount of locked tokens is decreased by the required amount

Unstaking all

This subscenario:

- Receives the following data from *Parent* or *Owner*:
    - query id

- The operation fails, if and only if:
    - The sender is not the *Parent* or the *Owner*
    - Fee can not be paid (standard)
- If the operation successes:
    - All the tokens are locked
    - Notification message is sent to the *Parent* with the following data:
        - query id
        - amount of tokens
        - *Owner*

Upgrade start

This subscenario:

- Receives:
    - query id
- The operation fails if and only if:
    - The sender is not *Parent*
    - Amount of *value* is not enough to cover fee
- In case of success:
    - *proxy_migrate_wallet* message to *Parent* is sent with the following data:
        - query id
        - amount of tokens
        - *Owner*
    - all current tokens are burnt

Upgrade completion

This subscenario:

- Receives the following data:
    - query id
    - amount of coins from the original wallet
- The operation fails if and only if:
    - The sender is not *Parent*
- In case of success:
    - Tokens equal to “amount of coins from the original wallet” are minted

Withdraw extra coins

It’s a debug subscenario, will not considered in the verification project

Token transfer initiated by Parent

It’s a debug subscenario, will not considered in the verification project

System activities

Bounce

This subscenario (should never happen):

- Receives the following data:
  - query id
  - operation code
  - possibly, additional data
- If the operation code is:
  - *receive_coins*
    - Always succeeds
    - Tokens amount is increased by the additional value provided
  - *proxy_reserve_tokens*
    - Always succeeds
    - The additional data represents a “locked value” and “locked value” is unlocked
  - *proxy_migrate_wallet*
    - Always succeeds
    - In case of success, the additional data represents “original tokens” and “original tokens” are minted
  - In all other cases nothing, nothing happens

Dispatching

This subscenario:

- Receives the following data:
  - Operation code
  - Bounce flag
  - Additional data
- If Bounce flag is on, Bounce scenario is executed
- Otherwise, if the Operation code is:
  - *send_tokens* - send_tokens scenario is executed
  - *receive_tokens* - receive_tokens scenario is executed
  - *tokens_minted* - tokens_minted scenario is executed
  - *save_coins* - save_coins scenario is executed
  - *unstake_tokens* - unstake_tokens scenario is executed
  - *rollback_unstake* - Rollback unstaking scenario is executed
  - *tokens_burned* - tokens_burned scenario is executed
  - *unstake_all* - Unstaking all scenario is executed
  - *upgrade_wallet* - Upgrade start scenario is executed
  - *merge_wallet* - Upgrade completion scenario is executed
  - Otherwise, the operation fails

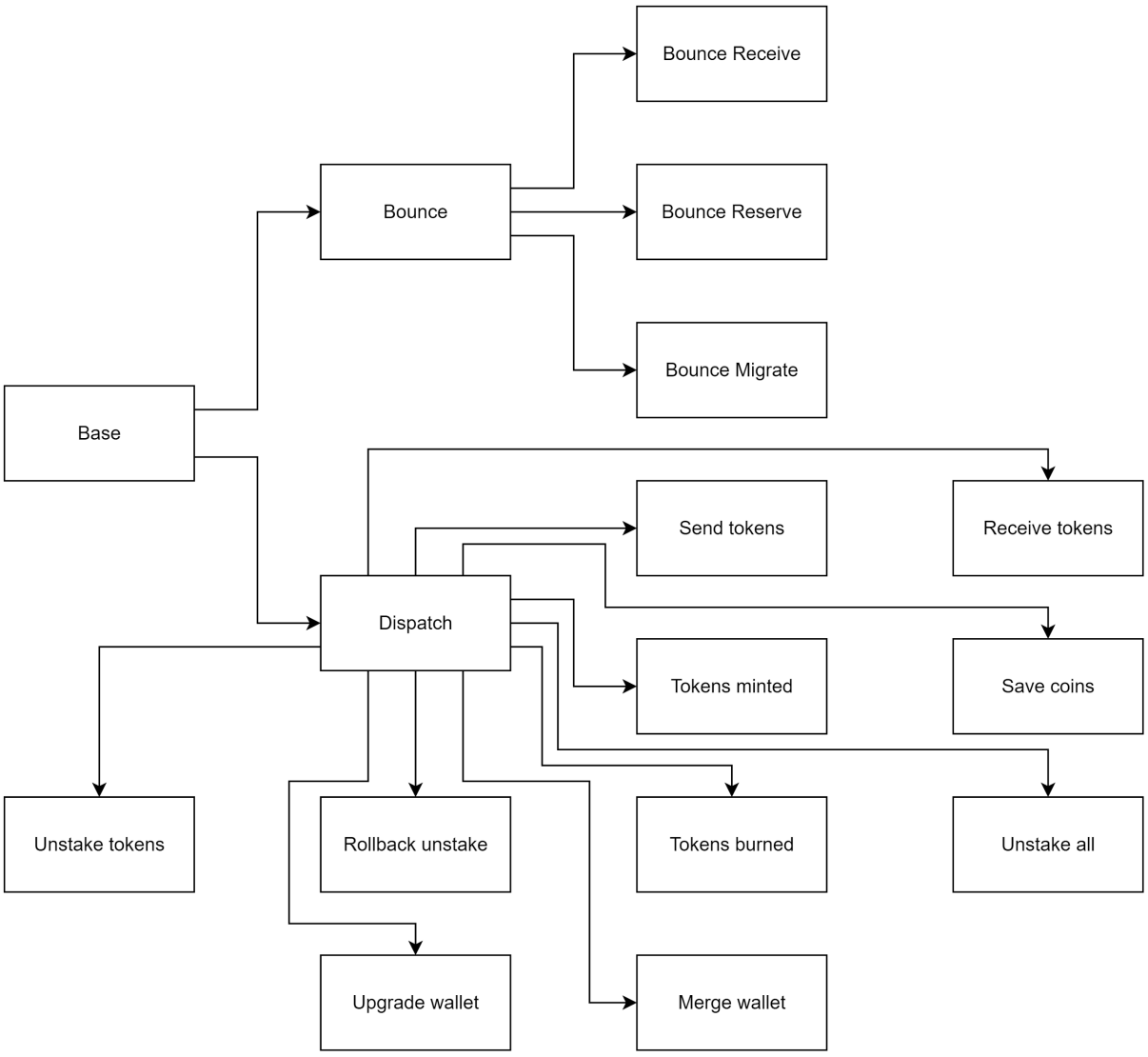
High-level specification

As it was mentioned above, the high-level specification is based on the following principles:

- Scenario-based specification
- High-level specification is independent from the implementation, while roughly follows it

- Has a multi-step model of moving from the less detailed to more detailed step as described above

The following scenarios and subscenarios were identified:



It's important to mention that this diagram demonstrates parent-children relationship rather than a call flow.

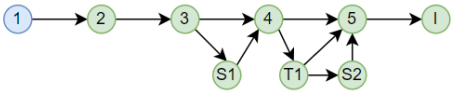
The details about each scenario and subscenario (and call flow, in particular), can be found at experimental web site https://ursus-tools.dev/3e3d7736-30fd-4717-9cec-b80aaebef64a.

Each subscenario there:

- Contains the following pages:
  - 1 - Textual scenario description
  - 2 - Output section
  - 3 - Input section

- S1 - States I
- 4 - Body section
- T1 - Types and Objects
- S2 - States II
- 5 - Details
- 6 - Invariants

- Navigation is performed using the elements with *green* background, where:
  - The main workflow is performed via ⇒ and ⇐ buttons (or using navigation graph):



- To go to the upper scenario or to the root pages click ⇧ button

All the Invariants from the last page are then considered by the Low-level Specification.

Low-level specification

Base Scenario

Input data:

in_msg_full:

flags	*src*	*rest*
uint4	address	slice

Statements:

N	Human description	Formal description
		$\forall flags \in uint4,\ src \in address, rest \in slice,\ s \in slice, \omega \in uint1, r = recv\_internal,$
		$b = on\_bounce,\ d = route\_internal\_message$
BAI.1	Anybody can initiate a message	$\exists f \in uint4, r \in slice, \sigma \in slice: \{\ \} r([f, src, r], \sigma) \{S = true\}$
BAI.2	The message is unhandled beforehand	Technical statement
	Otherwise, the result corresponds to:	

BAI.3	If bounced, to 'bounce' result	$\frac{\{flags\ \&\ 1\}r([flags,src,rest],s)\{S=\omega\}}{\{\ \}b(src,s)\{S=\omega\}}$
BAI.4	Otherwise, to 'dispatch' result	$\frac{\{flags\ \&\ 1=0\}r([flags,src,rest],s)\{S=\omega\}}{\{flags\ \&\ 1=0\}d(flags,src,s,rest)\{S=\omega\}}$
	If the operation successful, the outcome corresponds to:	
BAO.1	If bounced, to 'bounce' result	$\frac{\{flags\ \&\ 1\}r([flags,src,rest],s)\{S=true\}}{r([flags,src,rest],s)=b(src,s)}$
BAO.2	Otherwise, to 'dispatch' result	$\frac{\{flags\ \&\ 1=0\}r([flags,src,rest],s)\{S=true\}}{r([flags,src,rest],s)=d(flags,src,s,rest)\wedge(flags\ \&\ 1=0)}$

Bounce scenario

Input data:

s:

_	*op*	*query_id*	*amount*
uint32	uint32	uint64	coins

Statements:

N	Human description	Formal description
		$\forall\_ \in uint32,\ op \in uint32,\ query\_id \in uint64, amount \in coin,\ src \in slice, b = on\_bounce,$
		$c = route\_internal\_message$

BOI.1	Scenario can be initiated by 'Base' only	$\{caller \neq c\}b(src, [\_, op, query\_id, amount])\{S = false\})$
BOI.2	Message is unhandled beforehand	Technical statement
BOI.3	Otherwise, the operation result corresponds to the result of the underlying scenario	Merged with statements of underlying scenarios
BOI.4	Or, true, if no handler is found	$\{op \neq receive\_tokens \wedge op \neq proxy\_reserve\_tokens \wedge op \neq proxy\_migrate\_wallet\}$ $b(src, [\_, op, query\_id, amount])\{S = true\}$
	In case of success:	
BOO.1	Message is handled	Technical statement
BOO.2	The output ledger corresponds to the output ledger of the corresponding subscenario	Merged with statements of underlying scenarios

Bounce Receive Scenario

Input data:

s:

_	*op*	*query_id*	*amount*
uint32	uint32	uint64	coins

Statements:

N	Human description	Formal description
		$\forall \_ \in uint32,\ op \in uint32,\ query\_id \in uint64, amount \in coin,\ src \in slice, b = on\_bounce,$
		$op = receive\_tokens,\ c = route\_internal\_message$
BRI.1	Only 'Bounce' can initiate this scenario	Handled by BOI.1
BRI.2	In the sender is not an original sender, the scenario must fail	Unrelated, as bounce flag is automatically reset in case of improper sender
BRI.3	The state is 'Nothing' beforehand	Technical statement
BRI.4	Otherwise, the scenario is successful	$\{caller = c\}b(src, [\_, op, query\_id, amount])\{S = true\}$
	In case of success:	
BRO.1	Operation is handled afterwards	Technical statement
BRO.2	Amount of staked tokens is increased by message amount	$\frac{\{\ \}b(src,[\_,op,query\_id,amount])\{S=true\}}{\{\forall x: x=tokens\}b(src,[\_,op,query\_id,amount])\{tokens=x+amount\}}$

Bounce Reserve scenario

Input data:

s:

_	*op*	*query_id*	*amount*
uint32	uint32	uint64	coins

Statements:

N	Human description	Formal description
		$\forall \_ \in uint32,\ op \in uint32,\ query\_id \in uint64, amount \in coin,\ src \in slice, b = on\_bounce,$
		$op = proxy\_reserve\_tokens, c = route\_internal\_message$
BEI.1	Only 'Bounce' can initiate this scenario	Handled by BOI.1
BEI.2	In the sender is not a Parent, the scenario must fail	Automatically handled by TVM by resetting improper *bounce* flag
BEI.3	In case of too small amount of unstaked the scenario fails	$\{amount > unstaking\}b(src, [\_, op, query\_id, amount])\{S = false\}$
BEI.4	The state is 'Nothing' beforehand	Technical statement
BEI.5	Otherwise, the scenario is successful	$\{caller = c \wedge amount \leq unstaking\}b(src, [\_, op, query\_id, amount])\{S = true\}$
	In case of success:	
BEO.1	Operation is handled	Technical statement

	afterwards	
BEO.2	Amount of staked tokens is increased by message amount	$\frac{\{\ \}b(src,[\_,op,query\_id,amount])\{S=true\}}{\{\forall x{:}x=tokens\}b(src,[\_,op,query\_id,amount])\{tokens=x+amount\}}$
BEO.3	Amount of unstaked tokens is decreased by message amount	$\frac{\{\ \}b(src,[\_,op,query\_id,amount])\{S=true\}}{\{\forall x{:}x=unstaking\}b(src,[\_,op,query\_id,amount])\{unstaking=x-amount\}}$

Bounce Migrate scenario

Input data:

s:

_	*op*	*query_id*	*amount*
uint32	uint32	uint64	coins

Statements:

N	Human description	Formal description
		$\forall \_ \in uint32,\ op \in uint32,\ query\_id \in uint64, amount \in coin,\ src \in slice, b = on\_bounce,$
		$op = proxy\_reserve\_tokens, c = route\_internal\_message$
BMI.1	Only 'Bounce' can initiate this scenario	Handled by BOI.1
BMI.2	In the sender is not a Parent,	Automatically handled by TVM by resetting improper *bounce* flag

	the scenario must fail	
BMI.3	If the provided value is less than total amount of all rounds, scenario must fall	Covered by upper contracts
BMI.4	The state is 'Nothing' beforehand	Technical statement
BMI.5	Otherwise, the scenario is successful	$\{caller = c\}b(src, [\_, op, query\_id, amount])\{S = true\}$
	In case of success:	
BMO.1	Operation is handled afterwards	Technical statement
BMO.2	Amount of staked tokens is increased by message amount	
BMO.3	Amount of unstaked tokens is increased by message unstaked amount	Handled by upper contracts
BMO.4	Rounds are increased by message rounds	Handled by upper contracts

Dispatch scenario

Input data:

s:

op	*rest*
uint32	slice

rest$(op = 0)$

c	*rest*$_1$
uint8	slice

Statements:

N	Human description	Formal description
		$\forall\ op \in uint32,\ rest \in slice, flags \in uint4,\ src \in slice, cs \in slice, d = route\_internal\_message,$
		$r = receive\_tokens,\ \mu = tokens\_minted,\ \eta = save\_coins,\ u = unstake\_tokens$
		$\rho = rollback\_unstake,\ \beta = tokens\_burned,\ \nu = unstake\_all,\ \upsilon = upgrade\_wallet$
		$\omega = merge\_wallet,\ \sigma = send\_tokens$
		$\Sigma = \{r, \mu, \eta, u, \rho, \beta, \nu, \upsilon, \omega, \sigma\}$
		$f\ \&\ 1\ = 0,\ c = recv\_internal, \delta \in uint1$
DII.1	Scenario can be initiated by 'Base' only	$\{caller \neq c\}d(f, src, [op, rest], cs)\{S = false\})$
DII.2	Message is unhandled beforehand	Technical statement

DII.3	If the operation is unknown, the scenario fails	$\{op \notin \Sigma \wedge (op \neq 0 \vee (op = 0 \wedge rest.c \neq 85 \wedge rest.c \neq 117))\}$ $d(f, src, [op, rest], cs)\{S = false\}$
DII.4	Otherwise, the scenario result corresponds to the result of the underlying subscenario	Send tokens will be handled by the underlying scenario
		$\frac{\{op=r\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}r(src,rest)\{S=\delta\}}$
		$\frac{\{op=\mu\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}\mu(src,rest)\{S=\delta\}}$
		$\frac{\{op=\eta\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}\eta(src,rest)\{S=\delta\}}$
		$\frac{\{op=u\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}u(src,rest)\{S=\delta\}}$
		$\frac{\{op=\rho\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}\rho(src,rest)\{S=\delta\}}$
		$\frac{\{op=\beta\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}\beta(src,rest)\{S=\delta\}}$
		$\frac{\{op=\nu\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}\nu(src,rest)\{S=\delta\}}$
		$\frac{\{op=0\wedge(rest.c=85\vee rest.c=117)\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}\rho(src,"0000000000000000"s)\{S=\delta\}}$
		$\frac{\{op=\upsilon\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}\upsilon(src,rest)\{S=\delta\}}$
		$\frac{\{op=\omega\}d(f,src,[op,rest],cs)\{S=\delta\}}{\{\ \}\omega(src,rest)\{S=\delta\}}$

	In case of success:	
DIO.1	The message is handled	Technical statement
DIO.2	The output ledger corresponds to the output ledger of the corresponding subscenario	Send tokens will be handled by the underlying scenario
		$\frac{\{op=r\}d(f,src,[op,rest],cs)\{S=true\}}{op=r\Rightarrow d(f,src,[op,rest],cs)=r(src,rest)}$
		$\frac{\{op=\mu\}d(f,src,[op,rest],cs)\{S=true\}}{op=\mu\Rightarrow d(f,src,[op,rest],cs)=\mu(src,rest)}$
		$\frac{\{op=\eta\}d(f,src,[op,rest],cs)\{S=true\}}{op=\eta\Rightarrow d(f,src,[op,rest],cs)=\eta(src,rest)}$
		$\frac{\{op=u\}d(f,src,[op,rest],cs)\{S=true\}}{op=u\Rightarrow d(f,src,[op,rest],cs)=u(src,rest)}$
		$\frac{\{op=\rho\}d(f,src,[op,rest],cs)\{S=true\}}{op=\rho\Rightarrow d(f,src,[op,rest],cs)=\rho(src,rest)}$
		$\frac{\{op=\beta\}d(f,src,[op,rest],cs)\{S=true\}}{op=\beta\Rightarrow d(f,src,[op,rest],cs)=\beta(src,rest)}$
		$\frac{\{op=\nu\}d(f,src,[op,rest],cs)\{S=true\}}{op=\nu\Rightarrow d(f,src,[op,rest],cs)=\nu(src,rest)}$
		$\frac{\{op=0\}d(f,src,[op,rest],cs)\{S=true\}}{(op=0)\Rightarrow d(f,src,[op,rest],cs)=\nu(src,"0000000000000000"s)}$
		$\frac{\{op=\upsilon\}d(f,src,[op,rest],cs)\{S=true\}}{op=\upsilon\Rightarrow d(f,src,[op,rest],cs)=\nu(src,rest)}$
		$\frac{\{op=\omega\}d(f,src,[op,rest],cs)\{S=true\}}{op=\omega\Rightarrow d(f,src,[op,rest],cs)=\omega(src,rest)}$

Send Tokens scenario

Input data:

s(route_internal_message):

op	*rest*
uint32	slice

cs:

_1	*_2*	*_3*	*_4*	*fee*
address	coin	uint1	coin	coin

s(send_tokens):

query_id	*amount*	*recipient*	*return_excess*	*_5*	*forward_ton_amount*	*forward_payload*
uint64	coin	address	address	ref?	coin	slice

forward_payload:

_6	*rest_forward*
dict	slice

Output data:

o_name	*o_query_id*	*o_amount*	*o_owner*	*o_return_excess*	*o_forward_ton_amount*	*o_forward_payload*
uint32	uint64	coin	address	address	coin	slice

Statements:

N	Human description	Formal description
		$\forall\ op \in uint32,\ rest \in slice, flags \in uint4,\ src \in slice, s_1 \in slice, s_2 \in slice, cs \in slice$
		$_{-1} \in address, _{-2} \in coin, _{-3} \in uint1, _{-4} \in coin, fee \in coin$
		$query\_id \in uint64,\ amount \in coin,\ recipient \in address,\ return\_excess\ \in address$
		$_{-5} \in ref?, forward\_ton\_amount \in coin, forward\_payload \in slice$
		$_{-6} \in dict, rest\_forward \in slice$
		$forward\_payload = \left[_{-6}, rest\_payload\right]$
		$s_2 = \left[query\_id, amount, recipient, return\_excess, _{-5}, forward\_ton\_amount, forward\_payload\right]$
		$s_1 = [op, rest],\ cs = \left[_{-1}, _{-2}, _{-3}, _{-4}, _{-5}, fee\right]$

		$d = route\_internal\_message,\ \sigma = send\_tokens,\ f = get\_original\_fwd\_fee(fee, false)$
		$flags\ \&\ 1\ =\ 0,\ c = recv\_internal, op = send\_tokens$
		$f_f = f,\ f_c = send\_tokens\_fee()$
		$F = \left(forward\_ton\_amount = 0 \wedge msg.value \geq f_f + f_c\right) \vee$
		$\left(forward\_ton\_amount > 0 \wedge msg.value \geq 2f_f + f_c + forward\_ton\_amount\right)$
		$o \in Message,\ o\_name = receive\_tokens,\ o\_query\_id \in uint64, o\_amount \in coin$
		$o\_owner \in address,\ o\_return\_excess \in address, o\_forward\_ton\_amount \in coin$
		$o\_forward\_payload \in slice$
		$o.data = [o\_name, o\_query\_id, o\_amount, o\_owner, o\_return\_excess, o\_forward\_ton\_amount,$
		$o\_forward\_payload]$
STI.1	Only 'Dispatch' can initiate this scenario	Partially covered by DII.1
		$\{caller \neq d\}\sigma(src, s_2, f)\{S = false\}$
STI.2	If the sender is not the Owner, the scenario must fail	$\{src \neq owner\}\sigma(src, s_2, f)\{S = false\}$
STI.3	If the sender is recipient, the scenario must fail	$\{src = recipient\}\sigma(src, s_2, f)\{S = false\}$

STI.4	If the message value is insufficient, the scenario must fail	$\{\neg F\}\sigma(src, s_2, f)\{S = false\}$
STI.5	If the token amount is insufficient, the scenario must fail	$\{amount > tokens\}\sigma(src, s_2, f)\{S = false\}$
STI.6	The state is 'Nothing' beforehand	Technical statement
STI.7	Otherwise, the scenario is successful	$\{caller = d \wedge caller_1 = c \wedge src = owner \wedge src \neq recipient \wedge F \wedge amount \leq tokens\}$
		$\sigma(src, s_2, f)\{S = false\}$
	In case of success:	
STX.1	The inner function corresponds to the outer function	$\frac{\{\ \}d(flags,src,s_1,cs)\{S=true\}}{d(flags,src,s_1,cs)=\sigma(src,s_2,f)}$
STO.1	Operation is handled afterwards	Technical statement
STO.2	Out message is created and sent afterwards	Technical statement
STO.3	Amount of staked tokens is decreased by message amount	$\frac{\{\ \}\sigma(src,s_2,f)\{S=true\}}{\{\forall x: x=tokens\}\sigma(src,s_2,f)\{tokens=x-amount\}}$

STO.4	Out message is assigned	$\frac{\{\ \}\sigma(src,s_2,f)\{S=true\}}{\{\ \}\sigma(src,s_2,f)\{o\_query\_id=query\_id\}}$
		$\frac{\{\ \}\sigma(src,s_2,f)\{S=true\}}{\{\ \}\sigma(src,s_2,f)\{o\_owner=owner\}}$
		$\frac{\{\ \}\sigma(src,s_2,f)\{S=true\}}{\{\ \}\sigma(src,s_2,f)\{o\_amount=amount\}}$
		$\frac{\{\ \}\sigma(src,s_2,f)\{S=true\}}{\{\ \}\sigma(src,s_2,f)\{o.recipient=recipient\}}$
		$\frac{\{\ \}\sigma(src,s_2,f)\{S=true\}}{\{\ \}\sigma(src,s_2,f)\{o.value=forward\_ton\_amount\}}$
		$\frac{\{\ \}\sigma(src,s_2,f)\{S=true\}}{\{\ \}\sigma(src,s_2,f)\{o\_return\_excess=return\_excess\}}$
		$\frac{\{\ \}\sigma(src,s_2,f)\{S=true\}}{\{\ \}\sigma(src,s_2,f)\{o\_custom\_payload=custom\_payload\}}$

Receive Tokens scenario

Input data:

s:

query_id	*amount*	*sender*	*return_excess*	*forward_ton_amount*	*forward_payload*
uint64	coin	address	address	coin	slice

Output data:

o_name	*o_query_id*	*o_amount*	*o_sender*	*o_forward_payload*
uint32	uint64	coin	address	slice

Statements:

N	Human description	Formal description
		$\forall\ src \in slice, s \in slice,\ query\_id \in uint64, amount \in coin,\ sender \in address$
		$return\_excess \in address,\ forward\_ton\_amount \in address,\ forward\_payload \in slice$
		$o \in Message,\ o\_name = transfer\_notification,\ o\_query\_id \in uint64$
		$o\_amount \in coin,\ o\_sender \in address,\ o\_forward\_payload \in slice$
		$s = [query\_id, amount, sender, return\_excess, forward\_ton\_amount, forward\_payload]$
		$o.data = [o\_name, o\_query\_id, o\_amount, o\_sender, o\_forward\_payload]$
		$o.name = transfer\_notification,\ f = receive\_tokens,\ d = route\_internal\_message$
		$s_e = parse\_std\_addr(src),\ s_a = create\_wallet\_address(sender.to\_builder(),\ parent,\ my\_code())$
		$\Sigma = s_e.first = chain::base \wedge s_e.second = s_a.third$
RTI.1	Only 'Dispatch' can initiate this scenario	$\{caller \neq d\} f(src, s) \{S = false\}$

RTI.2	If the sender is not the expected, the scenario must fail	$\{\neg\Sigma\}f(src,s)\{S = false\}$
RTI.3	If the message value is insufficient, the scenario must fail	$\{msg.value < forward\_ton\_amount\}f(src,s)\{S = false\}$
RTI.4	Operation is not created beforehand	Technical statement
RTI.5	The state is 'Nothing' beforehand	Technical statement
RTI.6	Otherwise, the scenario is successful	$\{calller = d \wedge \Sigma \wedge msg.value \geq forward\_ton\_amount\}f(src,s)\{S = true\}$
	In case of success:	
RTO.1	Operation is handled afterwards	Technical statement
RTO.2	Out message is created and sent afterwards if, and only if, amount of forwarded coins is positive	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\neq\emptyset\Leftrightarrow forward\_ton\_amount>0\}}$
RTO.3	Amount of staked tokens is increased by message amount	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x: x=tokens\}f(src,s)\{tokens=x+amount\}}$
RTO.4	If out message is assigned, it's assigned to:	$\frac{\{\ \}f(src,s)\{S=true\wedge o\neq\emptyset\}}{\{\ \}f(src,s)\{o\_query\_id=query\_id\}}$

		$\frac{\{\ \}f(src,s)\{S=true \wedge o \neq \emptyset\}}{\{\ \}f(src,s)\{o.recipient=owner\}}$
		$\frac{\{\ \}f(src,s)\{S=true \wedge o \neq \emptyset\}}{\{\ \}f(src,s)\{o\_amount=amount\}}$
		$\frac{\{\ \}f(src,s)\{S=true \wedge o \neq \emptyset\}}{\{\ \}f(src,s)\{o.value=forward\_ton\_amount\}}$
		$\frac{\{\ \}f(src,s)\{S=true \wedge o \neq \emptyset\}}{\{\ \}f(src,s)\{o\_sender=sender\}}$
		$\frac{\{\ \}f(src,s)\{S=true \wedge o \neq \emptyset\}}{\{\ \}f(src,s)\{o\_forward\_payload=forward\_payload\}}$

Tokens minted scenario

Input data:

s:

query_id	*amount*	*coins*	*_*	*round_since*	*staking_coins*
uint64	coin	coin	address	uint32	coin

Output data:

o_name	*o_query_id*	*o_amount*	*o_owner*	*o_flag*
uint32	uint64	coin	address	uint1

Statements:

N	Human description	Formal description
		$\forall\ src \in slice, s \in slice,\ query\_id \in uint64, amount \in coin,\ coins \in coin$
		$\_ \in address,\ round\_since \in uint32,\ staking\_coins \in coin$
		$o \in Message,\ o\_name = transfer\_notification,\ o\_query\_id \in uint64$
		$o\_amount \in coin,\ o\_owner \in address,\ o\_flag \in uint1$
		$s = [query\_id, amount, coins, \_, round\_since, staking\_coins]$
		$o.data = [o\_name, o\_query\_id, o\_amount, o\_owner, o\_flag]$
		$o.name = transfer\_notification,\ f = tokens\_minted,\ d = route\_internal\_message$
		$o.flag = 0$
		$v(x) = staking\sim udict\_get(32,\ x)\sim load\_coins()$
MTI.1	Only 'Dispatch' can initiate this scenario	$\{caller \neq d\}f(src, s)\{S = false\}$
MTI.2	If the sender is not the Parent, the scenario must fail	$\{src \neq parent\}f(src, s)\{S = false\}$
MTI.3	If the amount of coins is zero, the scenario must fail	$\{coins = 0\}f(src, s)\{S = false\}$
MTI.4	If round is provided and amount of coins assigned for the round is insufficient, the scenario must fail	$\{round\_since > 0 \wedge v(round\_since) > coins\}f(src, s)\{S = false\}$

MTI.5	The state is 'Nothing' beforehand	Technical statement
MTI.6	Out message is not created beforehand	Technical statement
MTI.7	Otherwise, the scenario is successful	$\{caller = d \wedge src = parent \wedge coins > 0 \wedge (round\_since = 0 \vee v(round\_since) \leq coins)\}$
		$f(src, s)\{S = true\}$
	In case of success:	
MTO.1	Operation is handled afterwards	Technical statement
MTO.2	Out message is created and sent afterwards	Technical statement
MTO.3	Amount of staked tokens is increased by message amount	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x: x=tokens\}f(src,s)\{tokens=x+amount\}}$
MTO.4	In case a round is provided, its value is decreased by amount	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x: x=v(round\_since)\}f(src,s)\{v(round\_since)=x-amount\}}$
MTO.5	Out message is assigned	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_query\_id=query\_id\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.recipient=parent\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_amount=amount\}}$

		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.value=0\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_owner=owner\}}$

Save Coins scenario

Input data:

s:

query_id	*coins*	*_*	*round_since*	*staking_coins*
uint64	coin	address	uint32	coin

Statements:

N	Human description	Formal description
		$\forall\ src \in slice, s \in slice,\ query\_id \in uint64, coins \in coin$
		$\_ \in address,\ round\_since \in uint32,\ staking\_coins \in coin$
		$s = [query\_id, coins, \_, round\_since, staking\_coins]$
		$f = save\_coins,\ d = route\_internal\_message$
		$v(x) = staking\sim udict\_get(32,\ x)\sim load\_coins()$
SCI.1	Only 'Dispatch' can initiate this scenario	$\{caller \neq d\}f(src, s)\{S = false\}$

SCI.2	If the sender is not the Parent, the scenario must fail	$\{src \neq parent\}f(src,s)\{S = false\}$
SCI.3	The state is 'Nothing' beforehand	Technical statement
SCI.4	Otherwise, the scenario is successful	$\{caller = d \wedge src = parent\}f(src,s)\{S = true\}$
	In case of success:	
SCO,1	Operation is handled afterwards	Technical statement
SCO.2	Amount of saved coins for the particular round is increased by message amount	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x{:}x=v(round\_since)\}f(src,s)\{v(round\_since)=x+amount\}}$

Unstake Tokens scenario

Input data:

s:

query_id	*amount*	*return_excess*	*custom_payload*
uint64	coin	address	ref?

custom_payload:

mode	*ownership_assigned_amount*
uint4	coin

Output data:

o_name	*o_query_id*	*o_amount*	*o_owner*	*o_mode*	*o_ownership_assigned_amount*
uint32	uint64	coin	address	uint4	coin

Statements:

N	Human description	Formal description
		$\forall\ src \in slice, s \in slice,\ query\_id \in uint64, amount \in coin,\ return\_excess \in address$
		$custom\_payload \in ref?,\ mode \in uint4,\ ownership\_assigned\_amount \in coin$
		$o \in Message,\ o\_name = proxy\_reserve\_tokens,\ o\_query\_id \in uint64$
		$o\_amount \in coin,\ o\_owner \in address,\ o\_mode \in uint4$
		$o\_ownership\_assigned\_amount \in coin$
		$s = [query\_id, amount, return\_excess, custom\_payload]$
		$(custom\_payload = \varnothing \vee\ custom\_payload = [mode, ownership\_assigned\_amount])$
		$o.data = [o\_name, o\_query\_id, o\_amount, o\_owner, o\_mode, o\_ownership\_assigned\_amount]$
		$f = unstake\_tokens,\ d = route\_internal\_message$
		$M = custom\_payload?.mode\ ?:\ auto$

		$W = custom\_payload?.ownership\_assigned\_amount\ ?:\ 0$
		$f_u = unstake\_tokens\_fee()$
UTI.1	Only 'Dispatch' can initiate this scenario	$\{caller \neq d\}f(src, s)\{S = false\}$
UTI.2	If the sender is not the Owner and not the wallet itself, the scenario must fail	$\{src \neq owner \wedge src \neq this\}f(src, s)\{S = false\}$
UTI.3	If the message value is insufficient, the scenario must fail	$\{msg.value < W + f_u\}f(src, s)\{S = false\}$
UTI.4	If the token amount is insufficient or zero, the scenario must fail	$\{amount = 0 \vee amount > tokens\}f(src, s)\{S = false\}$
UTI.5	If the mode is incorrect, the scenario must fail	$\{M > 2\}f(src, s)\{S = false\}$
UTI.6	If 'return excess' is neither zero, nor 'Owner', the scenario must fail	$\{return\_excess \neq 0 \wedge return\_excess \neq owner\}f(src, s)\{S = false\}$
UTI.7	The state is 'Nothing' beforehand	Technical statement
UTI.8	Out message is not created beforehand	Technical statement

UTI.9	Otherwise, the scenario is successful	$\{caller = d \wedge (src = owner \vee src = this) \wedge msg.value \geq W + f_u \wedge amount > 0 \wedge$
		$amount \leq tokens \wedge M \leq 2 \wedge (return\_excess = 0 \vee return\_excess = owner)\}$
		$f(src, s)\{S = true\}$
	In case of success:	
UTO.1	Operation is handled afterwards	Technical statement
UTO.2	Out message is created and sent afterwards	Technical statement
UTO.3	Amount of staked tokens is decreased by message amount	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x: x=tokens\}f(src,s)\{tokens=x-amount\}}$
UTO.4	Amount of unstaked tokens is increased by message amount	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x: x=unstaking\}f(src,s)\{unstaking=x+amount\}}$
UTO.5	Out message is assigned	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_query\_id=query\_id\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.recipient=parent\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_amount=amount\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.value=0\}}$

		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_owner=owner\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_mode=M\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_ownership\_assigned\_amount=W\}}$

Rollback unstake scenario

Input data:

s:

query_id	*amount*
uint64	coin

Statements:

N	Human description	Formal description
		$\forall\ src \in slice, s \in slice,\ query\_id \in uint64, amount \in coin$
		$s = [query\_id, amount]$
		$f = rollback\_unstake,\ d = route\_internal\_message$
RUI.1	Only 'Dispatch' can initiate this scenario	$\{caller \neq d\}f(src, s)\{S = false\}$

RUI.2	If the sender is not the Parent, the scenario must fail	$\{src \neq parent\}f(src,s)\{S = false\}$
RUI.3	If amount of unstaked tokens is insufficient, the scenario must fail	$\{amount > unstaking\}f(src,s)\{S = false\}$
RUI.4	The state is 'Nothing' beforehand	Technical statement
RUI.5	Otherwise, the scenario is successful	$\{caller = d \wedge src = parent \wedge amount \leq unstaking\}f(src,s)\{S = true\}$
	In case of success:	
RUO.1	Operation is handled afterwards	Technical statement
RUO.2	Amount of staked tokens is increased by message amount	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x: x=tokens\}f(src,s)\{tokens=x+amount\}}$
RUO.3	Amount of unstaked tokens is decreased by message amount	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x: x=unstaking\}f(src,s)\{unstaking=x-amount\}}$

Tokens Burned scenario

Input data:

s:

query_id	*amount*	*coins*

uint64	coin	coin

Output data:

o_name	*o_query_id*	*o_amount*	*o_coins*
uint32	uint64	coin	address

Statements:

N	Human description	Formal description
		$\forall\ src \in slice, s \in slice,\ query\_id \in uint64, amount \in coin,\ coins \in coin$
		$o \in Message,\ o\_name = withdrawal\_notification,\ o\_query\_id \in uint64$
		$o\_amount \in coin,\ o\_coins \in coin$
		$s = [query\_id, amount, coins]$
		$o.data = [o\_name, o\_query\_id, o\_amount, o\_coins]$
		$f = unstake\_tokens,\ d = route\_internal\_message$
		$f_b = wallet\_storage\_fee()$
TBI.1	Only 'Dispatch' can initiate this scenario	$\{caller \neq d\} f(src, s) \{S = false\}$

TBI.2	If the sender is not the Owner and not the wallet itself, the scenario must fail	$\{src \neq parent\}f(src,s)\{S = false\}$
TBI.3	If the message value is insufficient, the scenario must fail	$\{msg.value < coins + f_b\}f(src,s)\{S = false\}$
TBI.4	If the token amount is insufficient, the scenario must fail	$\{amount > unstaking\}f(src,s)\{S = false\}$
TBI.5	The state is 'Nothing' beforehand	Technical statement
TBI.6	Otherwise, the scenario is successful	$\{caller = d \wedge src = parent \wedge msg.value \geq coins + f_b \wedge amount \leq unstaking\}$
		$f(src,s)\{S = true\}$
	In case of success:	
TBO,1	Operation is handled afterwards	Technical statement
TBO.2	Out message is created and sent afterwards	Technical statement
TBO.3	Amount of unstaked tokens is decreased by message amount	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x: x=unstaking\}f(src,s)\{unstaking=x-amount\}}$

TBO.5	Out message is assigned	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_query\_id=query\_id\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.recipient=owner\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_amount=amount\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.value=coins\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_owner=owner\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_coins=coins\}}$

Unstake All scenario

Input data:

s:

query_id
uint64

Output data:

o_name	*o_query_id*	*o_amount*	*o_owner*	*o_mode*	*o_ownership_assigned_amount*

uint32	uint64	coin	address	uint4	coin

Statements:

N	Human description	Formal description
		$\forall\, src \in slice, s \in slice,\ query\_id \in uint64$
		$o \in Message,\ o\_name = proxy\_reserve\_tokens,\ o\_query\_id \in uint64$
		$o\_amount \in coin,\ o\_owner \in address,\ o\_mode \in uint4$
		$o\_ownership\_assigned\_amount \in coin$
		$s = [query\_id]$
		$o.data = [o\_name, o\_query\_id, o\_amount, o\_owner, o\_mode, o\_ownership\_assigned\_amount]$
		$f = unstake\_all,\ d = route\_internal\_message$
		f_a - can not be calculated by the current implementation, the violation will lead to generic exception
UAI.1	Only 'Dispatch' can initiate this scenario	$\{caller \neq d\} f(src, s) \{S = false\}$
UAI.2	If the sender is neither the Owner nor the Parent, the scenario must fail	$\{src \neq owner \wedge src \neq parent\} f(src, s) \{S = false\}$
UAI.3	If the message value is	$\{msg.value < tokens + f_a\} f(src, s) \{S = false\}$

	insufficient, the scenario must fail	
UAI.4	If the token amount is insufficient or zero, the scenario must fail	$\{tokens = 0\}f(src, s)\{S = false\}$
UAI.5	The state is 'Nothing' beforehand	Technical statement
UAI.6	Otherwise, the scenario is successful	$\{caller = d \wedge (src = owner \vee src = parent) \wedge msg.value \geq tokens + f_a \wedge tokens > 0\}$
		$f(src, s)\{S = true\}$
	In case of success:	
UAO.1	Operation is handled afterwards	Technical statement
UAO.2	Out message is created and sent afterwards	Technical statement
UAO.3	Amount of staked tokens becomes zero	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{tokens=0\}}$
UAO.4	Amount of unstaked tokens is increased by amount of staked tokens	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x,y:x=unstaking,y=tokens\}f(src,s)\{unstaking=x+y\}}$
UAO.5	Out message is assigned	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_query\_id=query\_id\}}$

		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.recipient=parent\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x:x=tokens\}f(src,s)\{o\_amount=x\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.value=0\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_owner=owner\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_mode=auto\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_ownership\_assigned\_amount=0\}}$

Upgrade Wallet scenario

Input data:

s:

query_id
uint64

Output data:

o_name	*o_query_id*	*o_tokens*	*o_owner*
uint32	uint64	coin	address

Statements:

N	Human description	Formal description
		$\forall\ src \in slice, s \in slice,\ query\_id \in uint64$
		$o \in Message,\ o\_name = proxy\_migrate\_wallet,\ o\_query\_id \in uint64$
		$o\_tokens \in coin,\ o\_owner \in address$
		$s = [query\_id]$
		$o.data = [o\_name, o\_query\_id, o\_tokens, o\_owner]$
		$f = upgrade\_wallet,\ d = route\_internal\_message$
		$f_{\upsilon} = upgrade\_wallet\_fee()$
UWI.1	Only 'Dispatch' can initiate this scenario	$\{caller \neq d\} f(src, s)\{S = false\}$
UWI.2	If the sender is other than the Parent, the scenario must fail	$\{src \neq parent\} f(src, s)\{S = false\}$
UWI.3	If the message value is insufficient, the scenario must fail	$\{msg.value < f_{\upsilon}\} f(src, s)\{S = false\}$
UWI.4	The state is 'Nothing' beforehand	Technical statement

UWI.5	Otherwise, the scenario is successful	$\{caller = d \wedge src = parent \wedge msg.value \geq f_{\upsilon}\} f(src, s)\{S = true\}$
	In case of success:	
UWO.1	Operation is handled afterwards	Technical statement
UWO.2	Out message is created and sent afterwards	Technical statement
UWO.3	Amount of staked tokens becomes zero	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{tokens=0\}}$
UWO.4	Out message is assigned	$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_query\_id=query\_id\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.recipient=parent\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\forall x{:}x=tokens\}f(src,s)\{o\_amount=x\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o.value=0\}}$
		$\frac{\{\ \}f(src,s)\{S=true\}}{\{\ \}f(src,s)\{o\_owner=owner\}}$

Merge Wallet scenario

Input data:

s:

query_id	*new_tokens*
uint64	coin

Output data:

o_name	*o_query_id*
uint32	uint64

Statements:

N	Human description	Formal description
		$\forall\ src \in slice, s \in slice,\ query\_id \in uint64, new\_tokens \in coin$
		$o \in Message,\ o\_name = gas\_excess,\ o\_query\_id \in uint64$
		$s = [query\_id, new\_tokens]$
		$o.data = [o\_name, o\_query\_id]$
		$f = merge\_wallet,\ d = route\_internal\_message$
MWI.1	Only 'Dispatch' can initiate this scenario	$\{caller \neq d\} f(src, s) \{S = false\}$
MWI.2	If the sender is other than the Parent, the scenario must fail	$\{src \neq parent\} f(src, s) \{S = false\}$

MWI.3	The state is 'Nothing' beforehand	Technical statement
MWI.4	Otherwise, the scenario is successful	$\{caller = d \wedge src = parent\} f(src, s) \{S = true\}$
	In case of success:	
MWO.1	Operation is handled afterwards	Technical statement
MWO.2	Out message is created and sent afterwards	Technical statement
MWO.3	Amount of staked tokens is increased by message amount	$\frac{\{\ \} f(src,s) \{S=true\}}{\{\forall x: x=tokens\} f(src,s) \{tokens=x+amount\}}$
MWO.4	Out message is assigned	$\frac{\{\ \} f(src,s) \{S=true\}}{\{\ \} f(src,s) \{o\_query\_id=query\_id\}}$
		$\frac{\{\ \} f(src,s) \{S=true\}}{\{\ \} f(src,s) \{o.recipient=owner\}}$
		$\frac{\{\ \} f(src,s) \{S=true\}}{\{\ \} f(src,s) \{o.value=0\}}$

Verification

The verification code is placed at https://github.com/Pruvendo/hipo-wallet. All the assistance to:

- Get an access
- Understand the verification
- Privately run the verification
- Interpret the result and ensure the formal verification has really passed

will be provided by request, where Pruvendo engineers will explain:

- Transformation from Stage 6 to Stage 7 of the formal specification
- Conversion of the source code to Ursus
- Quickchicks

Please note, that to run the verification project you need:

- Ubuntu-based system (the latest available version is highly advised)
- Certain DevOps skills (while most of the instructions will be provided by the Pruvendo team)
- Rather powerful computer with at least:
    - Top-level set of CPU (such as a modern Xeon)
    - At least 32 GB of RAM (64 GB is preferred)
- Ability to wait up to 20 hours, until the whole process is completed

To run the formal verification by yourself, it's necessary to run the steps described in Appendix B.

Found issues and notes

The following suspicious points were found and reported to the developers:

N	Method	Description	Status
1	*onBounce*	No security checks	Handled by TVM itself, *bounce* bit is reset in case of improper handler
2	*tokens_minted*	no check if the amount is available (or even if the record exists)	Waived (covered by upper contracts)
3	*tokens_minted*	no check for positive number of coins	Waived (covered by upper contracts)
4	*save_coins*	no check for positive number of coins	Waived (covered by upper contracts)

5	*tokens_burned*	no check for enough tokens	Waived (covered by upper contracts)
6	*tokens_burned*	no check for enough coins (+fee)	Waived (covered by upper contracts)
7	*rollback_unstake*	no check amount is available	Waived (covered by upper contracts)
8	*upgrade_wallet*	unstaked or not yet minted tokens will be lost	Waived (covered by upper contracts)

Appendix A. Example of scenario system

Consider a travel reservation System. Here we can identify the following scenarios for two different groups of beneficiaries (of course, the real system would be much more complicated):

- Owner group
    - Make the system available
    - Get a report
    - Shutdown the project
- User group
    - Book a flight
    - Book a hotel
    - Book a car
    - Cancel a booking
    - Write to support

Note that, values “book a flight from Munich to Frankfurt” and “book a flight from London to Los Angeles” assume the same sequence of user actions so they correspond to the same scenario. Even more, in the real world the latter scenario will require some specific actions for the latter case (required by the US authorities), but it should be the same scenario.

Also, it should be a single scenario, say, for the different types of payment for the hotel - instant, or at the reception desk.

It’s also important to note that some sequences can be common for different scenarios - such as an instant payment that exists for all the three “booking” scenarios. At the later stages these common sequences should be moved to sub-scenarios, specified separately as standalone ones.

Appendix B: Running the formal verification project

It's important to mention that the only direct result of the formal verification is ***OK*** (or failure). So if someone wishes to fully verify the process, it's necessary to check all the verification processes beforehand. All these processes are described above and here the running process is described.

It's important to mention that the only direct result of the formal verification is ***OK*** (or failure). So if someone wishes to fully verify the process, it's necessary to check all the verification processes beforehand. All these processes are described above and here the running process is described.

At first, it is worth mentioning that a powerful machine and significant time frame is required. If these requirements are met, the following instruction must be followed.

1. Request access from Pruvendo
2. Wait for access granting
3. Clone the following *git* repositories:
    a. https://vcs.modus-ponens.com/ton/coq-elpi-mod/
    b. https://vcs.modus-ponens.com/ton/coq-finproof-base
    c. https://vcs.modus-ponens.com/ton/solidity-monadic-language
    d. https://vcs.modus-ponens.com/ton/pruvendo-base-lib
    e. https://vcs.modus-ponens.com/ton/ursus-standard-library
    f. https://vcs.modus-ponens.com/ton/pruvendo-ursus-tvm
    g. https://vcs.modus-ponens.com/ton/ursus-contract-creator
    h. https://vcs.modus-ponens.com/ton/ursus-proofs
    i. https://vcs.modus-ponens.com/ton/ursus-environment
    j. https://vcs.modus-ponens.com/ton/ursus-quickchick
4. Use following script to install them:

```
cd coq-elpi-mod && git checkout experimental
opam install --ignore-pin-depends -y .
cd ..
cd coq-finproof-base && git checkout master
opam install --ignore-pin-depends -y .
cd ..
cd solidity-monadic-language && git checkout master
opam install --ignore-pin-depends -y .
cd ..
cd pruvendo-base-lib && git checkout master
opam install --ignore-pin-depends -y .
cd ..
cd ursus-standard-library && git checkout master
opam install --ignore-pin-depends -y .
cd ..
cd pruvendo-ursus-tvm && git checkout master
```

```
opam install --ignore-pin-depends -y .
cd ..
cd ursus-contract-creator && git checkout main
opam install --ignore-pin-depends -y .
cd ..
cd ursus-proofs && git checkout main
opam install --ignore-pin-depends -y .
cd ..
cd ursus-environment && git checkout main
opam install --ignore-pin-depends -y .
cd ..
cd ursus-quickchick && git checkout main
opam install --ignore-pin-depends -y .
cd ..
```

5. Clone the project repository
6. Run dune b -j *<number of CPU cores>* (don't use all the available cores, otherwise operation system processes can become still)
7. Wait for an extended period of time (hours or even days)
8. Ensure, ***OK*** is received