Advanced Research & Development Center

Project S014
Automatic Formal Verification of Smart-Contracts

Flex System Functional Specification and Smart-Contracts Audit

Authors
Evgeny Shishkin
evgeny.shishkin@infotecs.ru

with assistance of
Sergey Tyurin (FreeTON Foundation)
Pavel Ivanov (Moscow State University)

Moscow
June, 2021

Contents

1 Introduction

The document consists of two independent parts. The first part defines the functional specification for Flex, a single implementation of Distributed Exchange for Free TON blockchain. The second part contains the smart-contracts program code reliability audit report.

The functional specification is written using the natural language, with very little mathematics, yet it is precise enough to be translated into formal statements within the chosen smart-contracts execution mathematical model and a specification language.

The program code audit was conducted using the code review approach, with sporadic testing in the blockchain development network.

During the code audit, we found at least 3 critical vulnerabilities. One of them represents the erroneous pattern, used in several places in the code. Besides, we found several others vulnerabilities that are of lower severity yet they have to be fixed before the system release. All vulnerabilities were presented to the authors of FLeX and received their acknowledgement.

1.1 Document Scope

In this document, we consider the Flex system, published in https://github.com/tonlabs/flex, commit record 283e6a89.

The scope of our work was limited by the following smart-contracts: Flex, FlexClient, Price, TradingPair, RootTokenContract, TONTokenWallet as was stated within the contest text.

We did not cover flexDebot, flexHelperDebot, PriceXchg, XchgPair, because they were not requested within the contest text.

2 FLEX System

In this section, we give a high-level overview of the system, together with its architecture description.

2.1 System Purpose

Users of Free TON blockchain are able to create different digital assests, such as NFT- and Fungible- tokens. If a token has a potential for monetization, the question of reliable token trading and token exchange arises.

Traditionally, this problem is solved by Distributed Exchange (DEX) systems. FLeX system is a single implementation of DEX for the Free TON blockchain. It is based on an Order Book processing.

FLeX let its users to buy, sell tokens for the native digital currency of FreeTON, or exchange one type of tokens for an other in a given ratio.

In our specification effort, we try to stay on a rather high level of abstraction, not to get drawn into the implementation details where it is not needed.

2.2 Terms

In the document, we use several terms that are defined below.

Term	Definition
Trading System	An Information System that helps organise assets trading or exchange between asset owners.
Decentralized Exchange (DEX)	A trading system implemented using a blockchain protocol. It gives extra guarantees such as transparency, censorship-resistance and operation durability for its users.
Token	A digital asset implemented in a form of smart-contract compatible with FlexToken standard.
User	A party that has an account within the blockchain. This party is able to transfer coins to other users, call smart-contract functions by sending messages and observing the blockchain state.
Token creator	A user with exclusive rights for the root token smart-contract operation. See FlexToken description for details.

Wallet owner	A user with exclusive rights for token wallet smart-contract operation.
Deal	An event of token exchange or token trade between two users.
Order	An intent to buy/sell/exchange tokens of the user or cancel previous order that is declared as a message sent into a Trading System.
Order direction	A direction of a trade: *buy* or *sell*.
Counter order	An order direction that is the opposite to the given one. For buy - it is sell, and vice versa. In case of exchange, an exchange order X/Y is a counter for the exchange order Y/X.
Trading Pair	A token that is compatible with FlexToken standard and that gets traded for Crystals in Flex.
Exchange Pair	A pair of tokens that are both compatible with Flex-Token standard and that could be exchanged one for the other in Flex.
Security	Sometimes we use this term interchangeably with Trading Pair term.

2.3 System Architecture

The whole system consists of a client (off-chain) and smart-contracts (on-chain) parts.

The principal scheme of the system is depicated in Fig.1
The system consist of two parts: Debot - a client program running on the client's computing device, sending messages into smart-contracts and retrieving data from the blockchain, and several smart-contracts: TradingPair, Price, Flex, FlexClient, FlexToken - these get created and operate within the blockchain.

Debot is a user interface for the system. It aggregates data for available trading pairs, forms an order book for trading pairs and assists the user in the orders management.

The TradingPair smart-contract contains data for some trading pair. A separate TradingPair smart-contract is created for each token traded in Flex.

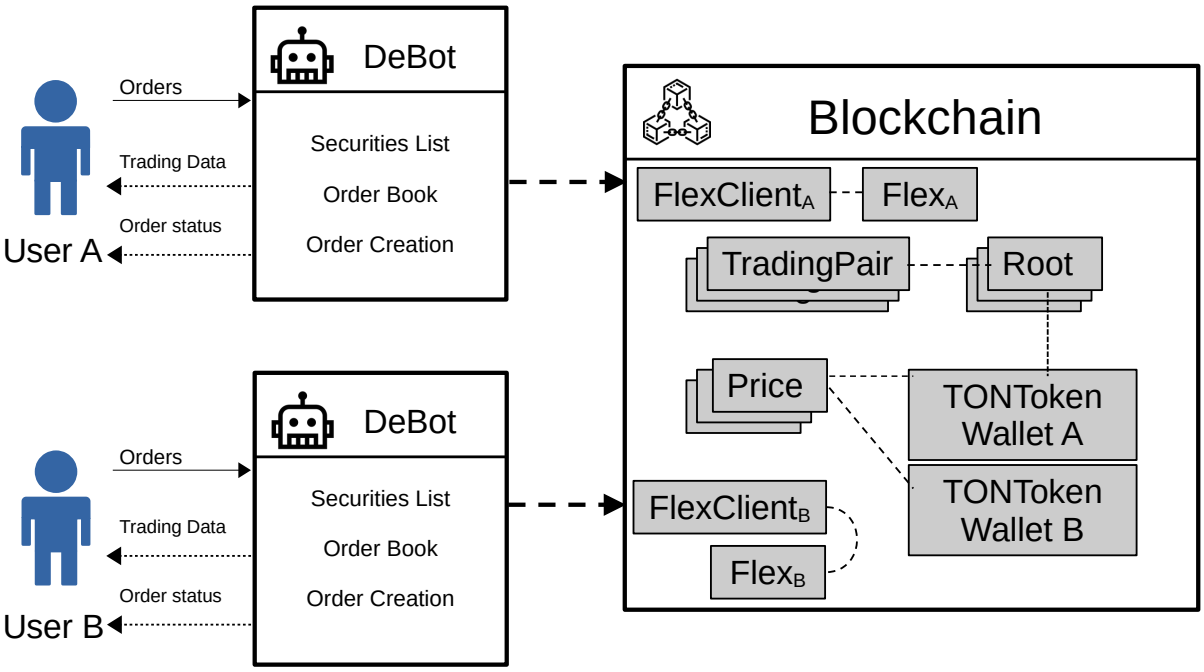

Figure 1: FLEX architecture

Each TradingPair corresponds to one and only one RootTokenContract smart-contract. The information regarding the token is stored in this contract (it is called Root on the scheme).

The Price smart-contract implements an order book for some trading pair at a fixed price level. It means that for different prices there will be a distinguished Price smart contract. The deal making logic (also known as the Matching Engine) is implemented there.

The TONTokenWallet smart-contract implements a user token wallet that are used to operate tokens during the trade or exchange.

The FlexClient smart-contract is a module implementing order creation, order cancelling and adding new trading pairs logic. It is a front-end that operate on other smart-contracts.

The Flex smart-contract contains some trading system options, such as commissions size.

2.4 Difference between FlexToken and TIP3

In the Free TON blockchain ecosystem, there is a known token standard called TIP3.

The token that is used in Flex has some difference in functionality comparing to TIP3, that is why we call it FlexToken despite the fact that it is mistakenly called TIP3 in the implementation code.

One of the major difference between the two is that FlexToken allows to temporarily promote some user into the token owner with limited capabilities.
This mechanism is needed to implement decentralized token transfer mechanism between the buyer and the seller wallets in case of a deal.

2.5 Usage Scenario

Here, we describe the main usage scenarios together with a little walkthrough on how it is implemented to shed some light on internals of FleX.
We have two users - a buyer and a seller - that would like to make a deal. Both users have wallets compatible with FlexToken.

The scenario of their interaction may look as follows:

1. Using DeBot, the buyer receive a list of trading pairs.

   The list of trading pairs is formed by scanning the blockchain state for the presence of smart-contracts with the given Code Hash. The hash should correspond to the hash of TradingPair code.

   For each TradingPair found, it reads the TIP3Root address. It then gets all relevant token information, such as: token description and token symbol from it.

   The diagram depicting this process is in Fig.3.

2. If a token found in the list, and DeBot receives the order book for this token.

   The order book consists of a set of values in a form $(price, buy, sell)$. It is a list of available buy and sell orders for different price levels.

The Order Book is formed by scanning the blockchain state for the presence of smart-contracts with a given Code Hash. The hash should correspond to the hash of Price code, constructed with parameters of TIP3Root address equal to TIP3Root address of the token under consideration.

DeBot retrives data of the from $(price, buy, sell)$ from each Price contract found this way. The data is a price in nano TONS per token, the amount of tokens available to be bought or sold. The diagram that depicts this process is in Fig.4.

3. The seller sends the order to sell their tokens, by putting the $price$ and $amount_1$ among other things, using their DeBot. The crucial thing here is that the seller lends ownership for his FlexToken wallet to the Price smart-contract at this step. Using this lending mechanism, Price will be able to trade tokens on behalf of the seller when the buyer comes up.

4. The buyer sends the order to buy the same token, by putting the $price$ and $amount_2$ among other things, using their DeBot. It equips its message with coins v that should be enough to make the deal, i.e. $v \geq price \times amount_2$

5. If prices of buy and sell match, both orders end up in the same Price smart-contract, in its order book.

6. The user counter orders will be matched and further processed either partly ($amount_1 \neq amount_2$) or in full ($amount_1 = amount_2$).

7. The buyer receives $amount_1$ tokens on his wallet, and the seller receives $amount_1 \times price$ coins.

8. After the deal is done for the seller, the Price contract returns ownership to the original wallet owner.

9. In case of partial matching, unprocessed part will be left in the order queue for further buying ($amount_2 > amount_1$) or selling ($amount_1 > amount_2$).

The diagram depicting the process of selling tokens is in Fig.5.

2.6 Flex vs. Other Exchanges

Usually, when someone mentions a Trading System, they assume a system that is able to reliably process buy, sell or exchange orders of different types.

Traditionally, there are several specific orders types available in such systems, such as: Market Order and Limit Order with an optional lifetime (GTC, FOK, etc).

Right now, there are only two types of orders available in Flex: the Limit Order *with a fixed price* to buy, sell, exchange tokens, with the Good-Till-Canceled lifetime, and a Cancel Order.
At the same time, the cancellation operation cancels all orders of the given direction, price level and token type. Partial cancellation is not supported right now.

Authors state that it is feasible to implement different types of orders, such as Market Order, using this architecture. At the moment of writing, this is not implemented, so we can not evaluate it.

2.7 User Capabilities

The end-user interacts with the system using the user interface implemented in Flex DeBot. This DeBot program aggregates trading data for the user and transfers its orders into the FlexClient contract.

The main enternace point of the system on the blockchain side is the FlexClient smart-contract. This smart-contract is able to perform the following actions:

1. Create trading or exchange pairs
2. Send buy, sell or exchange order
3. Cancel the orders
4. Transfer funds from the smart-contract to some chosen address
5. Get the system settings information

The API of this smart-contract is thoroughly described in 6.

3 Flex System Properties

In this section, we state the functional requirements that has to be met by different parts of Flex. Non-functional requirements, such as system performance, is not considered here.

3.1 System Layers Decomposition

To systematically cover the system with properties, we use the approach called *System Layers Decomposition*. We used this approach previously to formally specify and verify smart-contracts and it proved itself useful.

The approach is in the following:

The whole system gets separated into functional layers, see Fig.2. In this scheme, the layers located above rely on reliability of the layers located below.

Separation of the logic into layers gives us ability to specify and verify properties of different system parts either fully independently or above layer is verified with an assumption of reliability of all below layers. This approach considerably narrows the state space of the system that has to be checked.

The independence of different layers between each other could be checked formally. To do that, for each layer one has to write down all variables that are used to specify its behaviour in the formal specification - the initial set.
Next, we calculate a set of variables that are influenced by the variables from the initial set, for each layer. We call those sets as reachable sets. Two layers are indepented of each other if their reachable sets do not intersect.

If it turns out that reachable sets have intersections, we ensure that this intersection happens in a single assignment manner, meaning that the value from the layer below goes into the layer above strictly in one direction, once. In this case, if the layer below is considered reliable, the layer above it could be verified with this assumption.

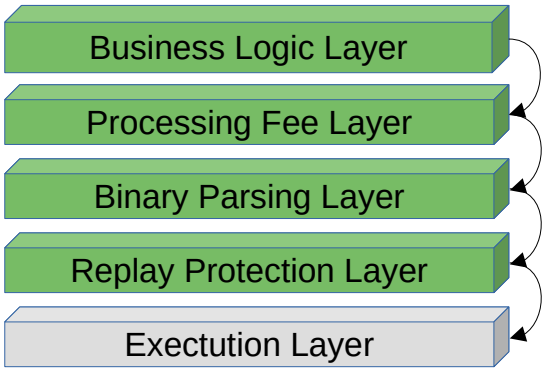

Figure 2: Layers of FLEX. Layers from above rely on reliability of layers from below. Green layers are to be verified. Grey layer is considered reliable.

3.2 Flex System Layers

We define the following layers for Flex:

1. **Business Logic Layer**. The main business logic layer of the system: orders management, trading pair management, order cancelation, order matching.

2. **Processing Fee Layer**. The layer that controls sufficiency of funds on smart-contract balance or user message value to process messages and store data within the blockchain.

3. **Binary Parsing Layer**. The logic of encoding and decoding the binary structures into structures of C++ programming language.

4. **Replay Protection Layer**. The logic protecting a smart-contract from processing of external previously sent messages.

5. **Execution Layer**. This layer contains all other functionality that has to be reliable to let the blockchain application function properly, including: the Virtual Machine, the blockchain protocol, networking protocol, blockchain node software, etc.

We now consider each layer, starting from the lowest one.

For each layer, we formulate statements in a natural language that must hold for this layer. *The link between those statements and the program code state variables are to be established on the formal specification and verification phase.*

3.3 Execution Layer

The whole system software stack required for running blockchain application goes into this layer.

1. C++ wrappers[1], that provide C++ TON primitives.
2. TON C++ Compiler
3. TVM Virtual Machine
4. Message delivery protocols Hypercube Routing, IHR
5. Distributed consensus protocol Catchain
6. TON Blockchain protocol
7. Blockchain node software (“the Node”)
8. Operation System and all relevant drivers
9. CPU and Network processors

We assume this whole stack to be reliable.

In particular, we rely on the following properties of the TON blockchain and TVM:

EXEC01. Internal messages between every pair of smart-contracts get delivered in the FIFO order, with the guarantee on delivery and uniqueness of delivery, when the IHR mode is turned off [2].

EXEC02. During delivery, some external messages from users to smart-contracts *may be lost*, but with repeated sending, eventually, external message gets delivered to the contract.

EXEC03. All delivered external messages are processed in FIFO order.

EXEC04. If a message gets delivered to the smart-contract inbound message queue, then, eventually, it will be processed by the smart-contract.

EXEC05. There is an upper bound on the number of permited computing operations allowed to be performed during processing of a single message, no matter how much coins you attach to it.

[1]Excluding *Replay Protection*, it is also provided in the form of wrapper

[2]see Durov - TVM Virtual Machine manual

3.4 Replay Protection Layer

This layer guarantees the uniqueness of external messages delivery. [3]

Uniqueness of delivery means that if an external message with the given body was previously delivered to the contract, the second delivery of the same message is either not allowed or allowed after some specified time period. [4].

Errors in this layer may lead to the violation of the whole system business logic. For example, it may be possible to repeatedly send previously sent buy order without actual user consent.

REPLAY01. If a message with the body b was delivered into the smart-contract once, then the next delivery of the same message b will be possible after no less than T seconds.

3.5 Binary Parsing Layer

In some FLeX smart-contracts, the authors resort to passing function input parameters in a form of data blob (so called bag of cells), with subsequent decoding it into programming language variables.

An error in this layer may lead to a complete distortion of data in contracts.

BINARY01. Encoding and decoding functions work correctly with respect to every possible permitted input, i.e. decoding some previously encoded value gives the same value for every possible input value.

3.6 Processing Fee Layer

Free TON blockchain collects fees for storing smart-contract code and data on a regular basis. That is why, to guarantee the durability of data stored inside the blockchain, we need to ensure that the balance of the contracts are always greater than some value needed to pay those fees.

Besides, each function call into the contract also has to be payed. The fee is collected either from the smart-contract balance, or from the coins that were attached to the message.

[3] The uniqueness of internal messages are guaranteed by the blockchain itself.

[4] If this interval is set big enough, almost all replay attacks become meaningless.

This layer specifies the balance of a smart-contract or of a message that is needed to process its business logic.

This layer IS NOT responsible for correct coins management that emerges during tokens buying or selling: the letter is not related to the platform fees.

Errors in this layer may lead to excessive coins loss, higher transaction processing times, and even deadlocks: it may be the case that smart-contract reach a state where no amount of attached tokens is enough to process its execution needs.

3.6.1 Storage Fee

Here we formulate basic storage fee properties.

STOR01. For smart-contracts with the fixed size of its state, such as FlexClient, TradingPair, XchgPair, Flex, RootTokenClient (if we do not change wallet_code), TONTokenWallet, it is required that they have got at least

$$storage\_fee(D) * (T_1 - T_2)$$

amount of coins at the end of each call, where $storage\_fee(D)$ - the function evaluates the amount of coins needed to be payed for keeping D code and data (in Kb) inside blockchain for 1 second, T_1 - expected life time of the contract, T_2 - actual life time of the contract starting from the deployment moment.

Note that T_2 value keeps monotonically increasing with each smart-contract call. Where T_1 is set during the deployment and may be increased afterwards.

STOR02. For smart-contracts that have variable state size, such as: Price, PriceXchg, they have to possess at least the following amount of coins after each call:

$$max\_storage\_fee(D) * (T_1 - T_2)$$

Here, $max\_storage\_fee(D)$ denotes the greatest amount of coins needed to store code and data D.

It may be problematic to calculate the exact upper bound for storage fee in the variable data size case. This is why we *over approximate* it with the greatest fee that has to be paid for maximum allowed data.

3.6.2 Execution Fee

We now consider fees that are collected during the execution phase of a message processing (also called as *a call*).

EXEC01. If a smart-contract have to pay for the call f, its balance must contain no less than:

$$processing\_fee(f, d)$$

Here, $processing\_fee(f, d)$ denotes the amount of coins needed cover fees for executing function f in a state d.
The $processing\_fee$ amount of coins cover fees that are taken for processing inbound messages, mere execution costs, outbound message forwarding fees.

EXEC02. If the call has to be paid from the coins attached to a message, its value has to be no less than:

$$processing\_fee(f, d)$$

Again, it may be problematic to calculate exact upper bound on fees that are needed to execute f in a state d. In this case, we can consider the following:

$$processing\_fee(D) = max\_processing\_fee(D)$$

Here, $max\_processing\_fee(f, d)$ denotes the amount of coins that guarantee any feasible execution of f. Please recall that executions in TVM are bounded, that is why this value is probably a constant.

3.6.3 Execution Limits

The TVM virtual machine allows to spend no more than G_{max} coins to pay for processing of a single call.

We have to guarantee that any call f with allowed parameters and reachable state variables will spend no more than G_{max} amount of coins during its execution. Otherwise, the smart-contract may step into a deadlock state: no amount of coins could help it make any further progress.

LIM01. For each smart-contract function f, for any tuple of valid arguments p, for every reachable state s, the execution cost of $f(p)$ within the state s takes no more than G_{max} coins to execute.

3.7 Business Logic Layer

In this layer, we focus on the higher business logic of the system, without considering low level details of other layers discussed above. [5]

The front-end API of the system is placed into two modules: FlexClient and FlexToken (consists of two smart-contracts).
Let us consider each of them.

BUS01. The trading pair management and order management calls are processed only if they are cryptographically signed with the private key of the FlexClient contract's owner. Otherwise, the call is aborted with the error *message_sender_is_not_my_owner*. This property holds for all calls from sections 3.7.1, 3.7.2

3.7.1 Trading Pair Management

In this section we state the functional capabilities of a FLeX user regarding the Trading Pair management. Here, the term *request* denotes an external message that gets sent into the FlexClient smart-contract by the user.

SEC01. If a user sends request to create a Trading Pair P attaching M coins to their request, then the trading pair P will be added into the securities list and the user will be provided the identifier of this trading pair. The following inequality must hold

$$M \geq DeployValue$$

where $DeployValue$ is a special parameter that is provided into the Trading Pair depending on the life time chosen for this trading pair.

[5] Note that this approach allows us to be both precise and concise. By doing it this way, we avoid having huge incomprehensible specifications where all details are put into a single statement.

SEC02. If an error happens during the trading pair creation, the user is provided with the error code and the remaining coins initially sent with the request.

SEC03. For every token that is traded, there must be only one item in the security list.

SEC04. The securities list can change over time: trading pairs may come and go, depending on their life time.

3.7.2 Order Management

In this section we state the functional capabilities of a FLeX user regarding the buy/sell/exchange orders management. The phrase *send the order* denotes a request containing the order that is sent into the FlexClient smart-contract.

ORD01. If a user sends an order to buy (or sell) n tokens with the price p and a lifetime T, supplying all necessary parameters, then such an order will either be processed immediately or will be placed in the queue after partial processing or will be placed in the queue without processing.

ORD02. The order gets placed into the queue without any fulfilment if, at the moment of processing, there were no counter orders in the queue that could match it.

ORD03. For the order placed inside the queue, if the counter order will emerge within the time frame of T, then the original order will be matched and processed into a deal, partially or fully.

ORD04. For the order placed inside the queue, if no counter order emerges within the time frame of T, then there will be no deal with this order. It will expire and get removed from the queue.

ORD05. If an order gets fully matched and processed into a deal, then the buyer receives n tokens on his FlexToken wallet, and the seller receives $n \times v$ coins on his FlexToken wallet.

ORD06. If the order gets partially matched, then the buyer will receive n_1 tokens on his wallet, where $n_1 < n$ - the matched part of the original order, and the seller receives $n_1 \times v$ coins on his wallet.

ORD07. In case of a partial deal, the unprocessed order part with an amount $n - n_1$ stays inside the queue, awaiting counter orders.

ORD08. In case the buy order is sent, the user temporarily lends ownership to the corresponding Price smart-contract. It is guaranteed that the lending will expire within $lend\_finish\_time$ seconds, and the ownership will return to the user within $lend\_finish\_time + \delta$ seconds. Here, δ denotes the time needed to process messages sent to return the ownership to the owner.

ORD09. Trading orders of the same direction that are delivered into the system get processed using the FIFO ordering.

ORD10. Upon sending the cancel order request, there are several possibilities:

- All buy (or sell) user orders with the specified price level will be canceled.
- Some buy (or sell) user orders will be processed, and unprocessed part will be canceled.
- All buy (or sell) orders with the specified price level will be cancelled

The outcome of this operation depends on the counter orders emergence within the order queue before the cancel order gets delivered into the system.

Please note that the user is not able to cancel their order partially..

ORD11. The user is always able to transfer the remaining coins from FlexClient smart-contract to any other address.

3.7.3 Tokens Management

Here we consider a FlexToken-compatible token smart-contract that is composed of two parts: the token creator part - RootTokenContract and a user wallet smart-contract - TONTokenWallet.

TOK01. The user is always able to create their own token. After the creation, the user becomes the *token creator*. To create a token, user specifies the following parameters: token name, token symbol, decimals, and total emission. After the token gets created, the token creator receives its unique identifier.

TOK02. For every ordered collection of parameters

1. name
2. symbol
3. decimals
4. totalSupply
5. rootPubKey
6. rootAddr

there is always not more than 1 root token contract RootTokenContract available.

TOK03. The token creator is always able to add a token wallet for some user together with nominating it with some amount of tokens, unless all emitted tokens are already nominated. The user that is a receiver of the wallet becomes *the wallet owner*.

The token creator specifies the amount of tokens to be nominated to the newly created wallet and amount of coins that get transfered to the wallet. As a result, the token creator receives an identifier for the newly created wallet, and the user - wallet owner - obtains access to his wallet.

TOK04. Token creator is always able to create new wallet for a user without nominating him any tokens.

TOK05. A user is always able to create an empty wallet for the token of his choice. They have to provide address for its root contract in this case. Tokens may be received only from token creator contract or other token wallets.

TOK06. Token creator is always able to nominate some tokens to token wallet if the following inequality holds:

$$TotalGranted + T \leq TotalSupply$$

where $TotalGranted$ - the number of tokens already nominated to other wallets, $TotalSupply$ - the total tokens emission.

TOK07. The following inequality always hold for the Root token:

$$totalGranted \leq totalSupply$$

$$totalSupply \geq \sum_{w \in Wallets} getBalance(w)$$

where $Wallets$ - a set of existing token wallets

3.7.4 Wallets Management

The term *token wallet* denotes the TONTokenWallet smart-contract of FlexToken.

WAL01. Any mutating token operations are permitted only for the token owner or the token wallet lender if the lending period is not expired.

WAL02. The wallet lender has limited capabilities. In particular, they are not able to set other lenders, to transfer tokens above the lending amount, to change the lending period.

WAL03. The wallet owner is always able to transfer v tokens from their wallet to some other wallet of the same token if its balance is not less than v tokens. The destination wallet will receive exactly v tokens afterwards.

WAL04. The wallet owner is always able to delete their wallet if it contains 0 tokens. In this case, the remaining coins get transfered from the wallet balance to the balance of the specified address. After the wallet is deleted, it is no longer available to the wallet owner.

WAL05. The wallet owner W is able to lend their ownership to some other user by providing the user identifier U - the lender, lending amount of tokens V, lending time interval T.
At the same time, it is guaranteed that:

1. The wallet management will not be available for the lender after the interval T has elapsed.
2. The lender U may transfer tokens of amount not greater than V
3. The lender U is not able to set another lender
4. The lender U is not able to prolong the lending period T

5. The wallet owner W is not allowed to operate on their wallet until the lending period expires or the lender deliberately returns the rights to the original owner.

WAL06. The wallet owner is not able to prematurely stop the lender ownership before his lending period expires.

3.7.5 Orders Matching

The FleX system has its own matching engine mechanism. It operates in the following way: when the order gets delivered into the system, it is put in the speically distinguished queue, for each order direction there is a separate queue. Next, the matching engine gets executed. It matches orders of opposite directions with each other producing deals.

We now state some requirements for the Matching Engine to ensure its operation reliability.

MTE01. In case of successful matching of the counter orders into a full deal, both the buyer and the seller get notified about this event. If the order is matched only partially, the corresponding party will not receive the notification.

MTE02. The deal notification is sent exactly once.

MTE03. The order has a life time that is defined by the lending period field. The orders with expired lending period will not be matched.

MTE04. The full order may be matched into the deal no more than once. Partial matching may happen several times.

MTE05. If the order is put into the queue, but never matched despite the incoming counter orders, it means that its life time has expired or the user cancelled the order.

4 Sequence Diagrams

In this section, we present several UML-sequence diagrams that shed some light on how different system components interact in different usage scenarios.

We assume that each user has a FlexToken wallet.

4.1 Securities List Aggregation

To work with the Trading System, the user has to be equipped with the securities list and order books.
How securities list gets aggregated is depicted on Fig.3

4.2 Order Book Aggregation

In the previous step, the trading pairs were conducted together with corresponding RootTokenContract addresses (which we call tip3Addr here). Having this in hand, the order book is conducted as depicted in Fig. 4

4.3 Sell Order Processing

Sell order is processed as depicted on Fig.5

4.4 Buy Order Processing

Buy order is processed as depicted on Fig.6

4.5 Cancel Order Processing

All user buy (or sell) orders of a given price get cancelled with this request.

The request is processed as depicted on Fig.7

5 Code Audit Report

The program code audit was conducted using the code review approach, with sporadic testing in the blockchain development network.

The list of found vulnerabilities is presented below. All of them were confirmed by the FleX authors.

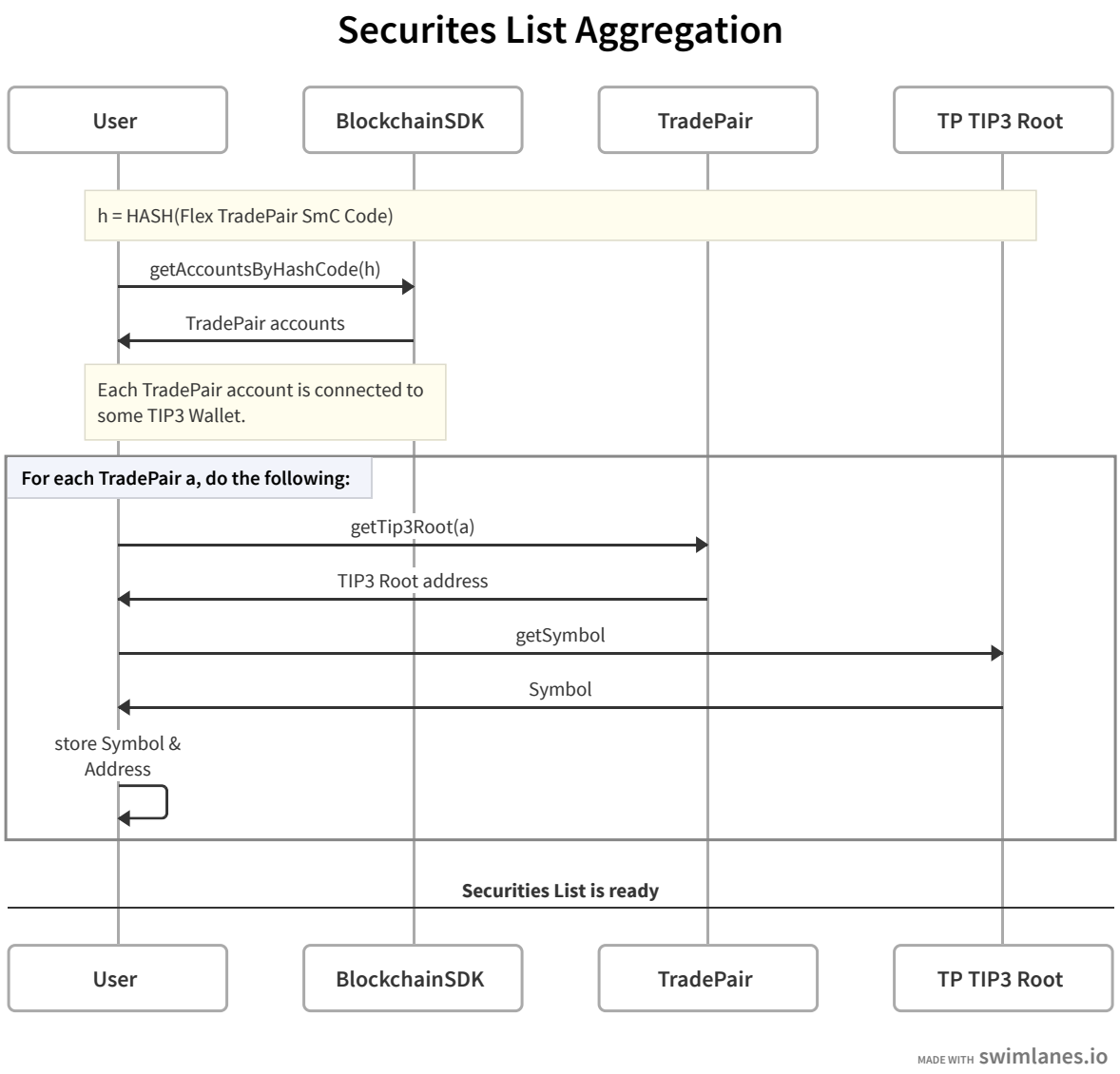

Figure 3: Securities list aggregation.

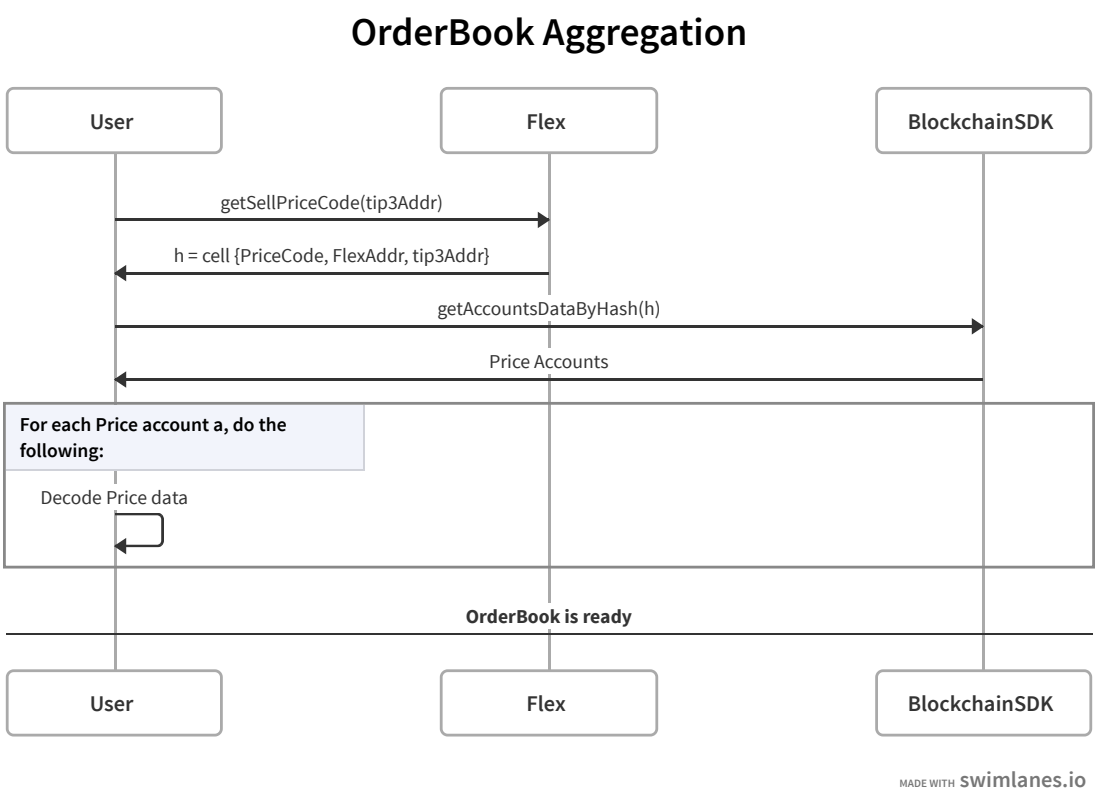

Figure 4: Order book aggregation for a trading pair.

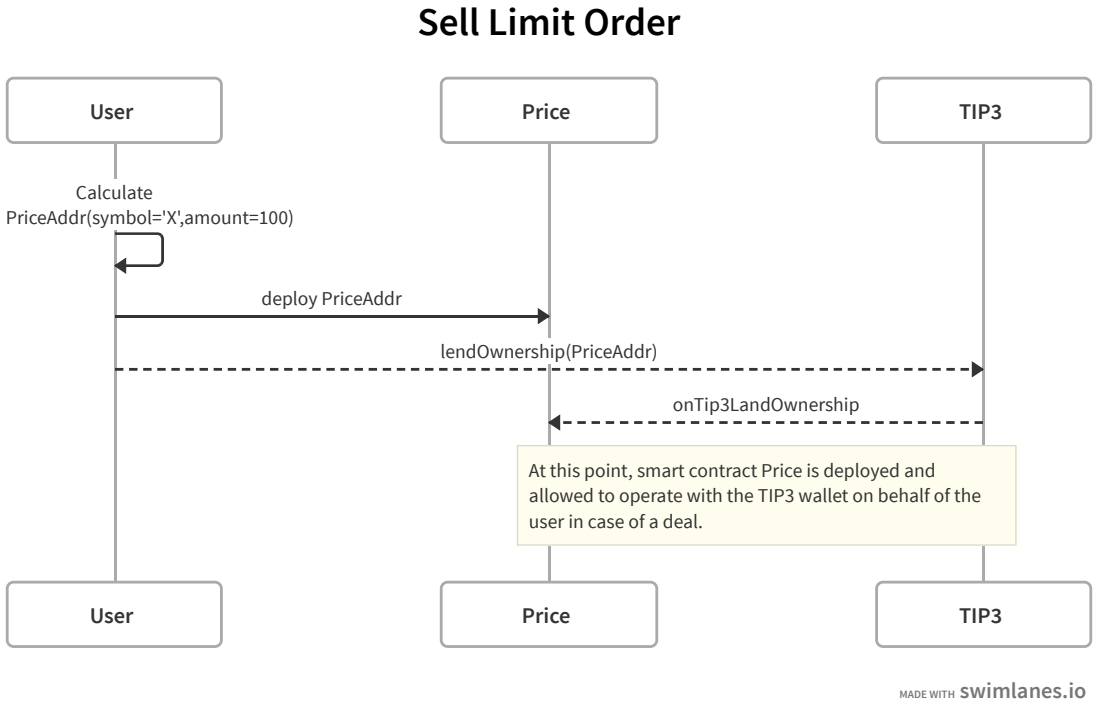

Figure 5: Sell order processing diagram

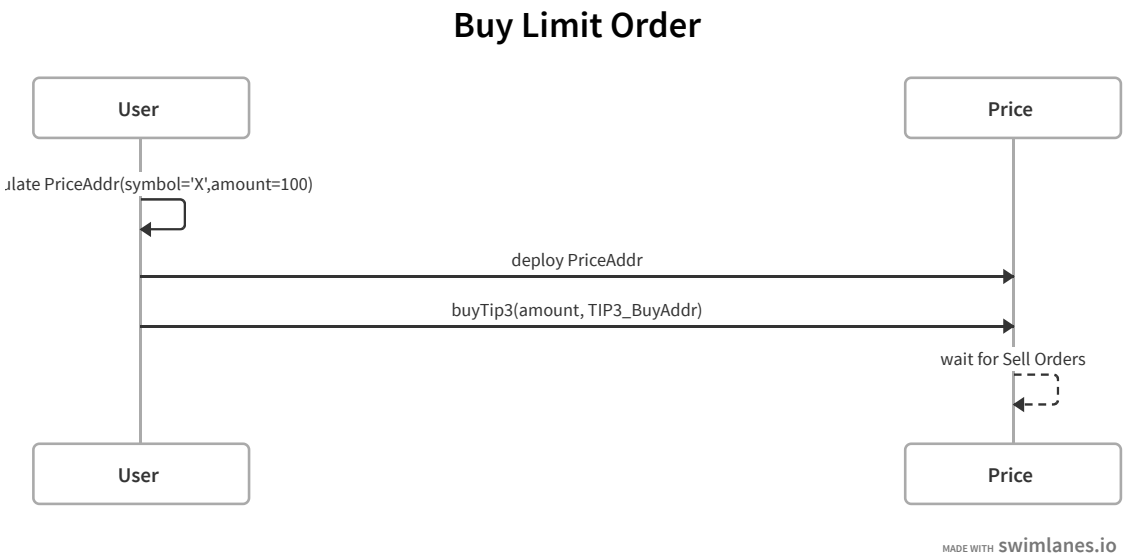

Figure 6: Buy order processing diagram

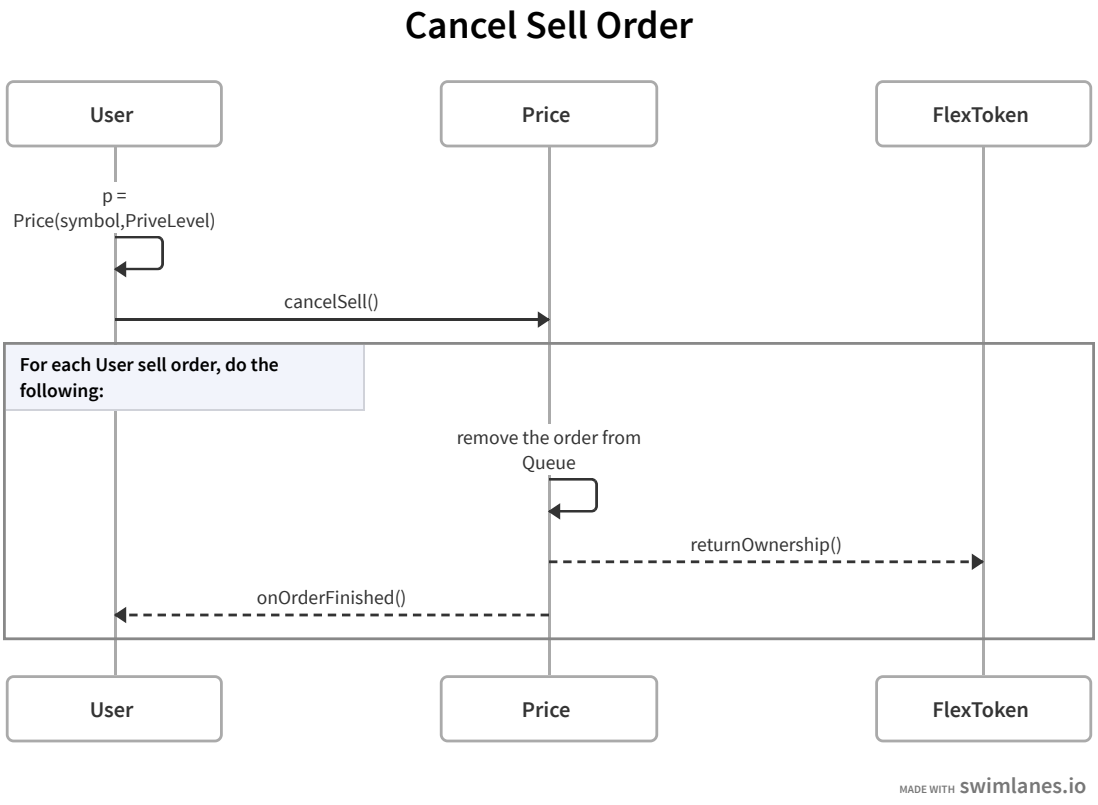

Figure 7: Cancel order processing diagram.

5.1 Wallet Code Replacement

The following method in *RootTokenContract* allows to set the wallet code.

```
void setWalletCode(cell wallet_code)
```

It was added as a separate entity in order to facilitate the constructor in terms of gas usage.

A check preventing from reinstallation of the code is erroneously commented out. Due to this fact, token creator can arbitrarily change the program code of user's wallets, and, potentially, issuing incompatible tokens.

5.2 Price Queue Overflow

Users orders get into the Price smart contract. They are stored in two queues there: one is for buy orders, the other is for sell order.

```
struct DPrice {
 // ...
 queue<OrderInfo> sells_;
 queue<OrderInfo> buys_;
};
```

There may be a scenario when one of the queues become bigger than the smart contract could store. This scenario is not processed in any way.

It would be possible to specify an upper limit on the number of orders in the queue, and return an error if the queue is full. So, at least, the contract will not enter some an undefined state.

5.3 Wallet Deployment Error

There is a method in *RootTokenContract* contract that creates a new wallet for the user with a simultaneous nomination of tokens.

```
  address deployWallet(int8 workchain_id,
                       uint256 pubkey,
                       uint256 internal_owner,
                       TokensType tokens,
                       WalletGramsType grams)
```

The *tokens* tokens are nominated to the user's wallet at the moment of creation wherein the amount of *reserved* tokens $totalGranted$ increases by the same number.

The deployment of the wallet as well as the simultaneous accept method is called for the newly created wallet. This call forms a message send action and requires some significant amount of gas.

The standard flags for sending a message include the option *IGNORE_ACTION_ERRORS* in this case. It means that if there is not enough funds on the root contract balance, then the transaction will be considered successful anyway.

It leads to the case when the number of issued tokens *total_granted* will be incorrectly increased by *tokens*, while the user wallet receives nothing.

Similar error is also found at the RootTokenWallet.cpp:108.

5.4 Unlimited Tokens Emission

The *mint* method is implemented in the smart-contract RootTokenWallet. It allows for additional, not limited by any constraint, emission of tokens which leads to its devaluation.

```
  void mint(TokensType tokens)
```

In our discussion with the author, we found out that this code is a template and users are expected to adjust it to their business logic. But we didn't find any mentioning of this, and it is not obvious for us because, despite this method, the whole token contract seems to be self-sufficient.

In our opinion, this code should be removed.

5.5 Trading Pairs Duplicate

To introduce a trading pair, a user calls the function

```
address deployTradingPair(address tip3_root,
                          uint128 deploy_min_value,
                          uint128 deploy_value);
```

In this case, the smart contract TradingPair is created. It consists of such variables:

```
struct DTradingPair {
 address flex_addr_;
 address tip3_root_;
 uint128 deploy_value_;
}
```

By specifying the same values $flex\_addr\_$ and $tip3\_root\_$ for different values of $deploy\_value\_$ the list of traded pairs can be filled with a lot of duplicates. These duplicates will point to the same token, but, at the same time, there will be a redundant list of traded tokens on the client side.

6 FlexToken Smart Contract. API Description

Let us take a closer look at each FlexToken function.

6.1 Trading Pair Creation

A trading pair is understood as a unique binding of the RootTokenContract address and the trade system settings, stored in FlexAddress with which this token is ready to work with.

```
address deployTradingPair(address tip3_root,
                          uint128 deploy_min_value,
                          uint128 deploy_value);
```

Method parameters:

1. tip3_root - A token root contract address
2. deploy_min_value - A number of coins reserved on the balance of TradingPair smart contract to ensure payment of storage fees on the blockchain
3. deploy_value - A number of coins the user attached

Limitations:

1. Method call is possible only by external message
2. The message must be signed with the owner's key

Return value:

- The function returns the address of the smart contract of the created pair

Possible errors:

- message_sender_is_not_owner the message is not signed by the owner of the contract
- not_enough_tons - the attached coins are not enough to deploy a pair

6.2 Exchange Pair Creation

Deploy Exchange Pair contract. It contains the root token contract addresses of the exchange pair (A - called major root / B - called minor root).

```
address deployXchgPair(address tip3_major_root,
                       address tip3_minor_root,
                       uint128 deploy_min_value,
                       uint128 deploy_value)
```

Method parameters:

1. tip3_major_root - Major Token Currency Root contract address
2. tip3_minor_root - Minor Token Currency Root contract address
3. deploy_min_value - The number of coins reserved on the balance of the pair's contract
4. deploy_value - The number of coins the user attached

Limitations:

1. Method call is possible only by an external message
2. The message must be signed with the owner's key

Return value:

- The function returns the smart contract address of the created pair

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract
- not_enough_tons - the attached coins are not enough to deploy a pair

6.3 Sell Order Placement

A sell order is formalized in the form of smart contract Price with specified parameters.

```
address deployPriceWithSell(cell args_cl)
```

Method parameters encoded in cell args_cl:

1. price - A token sale price measured in nano TON
2. amount - An amount of tokens for sale
3. lend_finish_time - A time given to complete a transaction. After this time the order is considered to be expired and removed from the queue.
4. min_amount - The minimum number of tokens the user is willing to sell
5. deals_limit - A maximum number of deals that could be matched in one call to the the matching engine
6. tons_value - A number of coins transferred to the contract Price
7. price_code - Price.tvc smart contract code in the Base64 encoding
8. addrs - reference to the address structure of TIP3 token currency send and receive wallets
9. tip3_code - TIP3 Wallet smart contract code in the Base64 encoding
10. tip3cfg - a reference to the structure of TIP3 Root configuration (name, symbol, decimals, root_public_key, root_address)

Limitations:

1. Method call is possible only by external message
2. The message must be signed with the owner's key

Return value:

- The function returns the address of the pair's smart contract.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract
- not_enough_balance

6.4 Buy Order Placement

A sell order is formalized in the form of smart contract Price with specified parameters.

```
address deployPriceWithBuy(cell args_cl)
```

Method parameters encoded in cell args_cl:

1. price - A token sale price
2. amount - An amount of tokens for sale
3. order_finish_time - A time given to complete an order. After this time the order is considered to be expired and removed from the queue.
4. min_amount - A minimum number of tokens the user is willing to trade
5. deals_limit - A maximum number of transactions for an instrument that can be matched in one cycle of the Order Execution Machine
6. tons_value - An amount of TONs to deploy the contract
7. price_code - Price.tvc smart contract code in the Base64 encoding
8. my_tip3_addr - An address of TIP3 wallet
9. tip3_code - TIP3 Wallet smart contract code in the Base64 encoding
10. ref<Tip3Config> tip3cfg - a reference to the structure of TIP3 Root configuration (name, symbol, decimals, root_public_key, root_address)

Limitations:

1. Method call is possible only by external message

2. The message must be signed with the owner's key

Return value:

- The function returns the address of the pair's smart contract.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract
- not_enough_tokens_amount -
- too_big_tokens_amount
- not_enough_tons_to_process

6.5 Sell Order Cancellation

Cancel the order to sell tokens.

```
void cancelSellOrder(cell args_cl)
```

Method parameters encoded in cell args_cl:

1. price - A token sale price
2. min_amount - A minimum number of tokens the user is willing to sell
3. deals_limit - A maximum number of transactions for an instrument that can be matched in one cycle of the Order Execution Machine
4. value - An amount of TONs to deploy the contract
5. price_code - Price.tvc smart contract code in the Base64 encoding
6. tip3_code - TIP3 Wallet smart contract code in the Base64 encoding
7. ref<Tip3Config> tip3cfg - a reference to the structure of TIP3 Root configuration (name, symbol, decimals, root_public_key, root_address)

Limitations:

1. Method call is possible only by external message
2. The message must be signed with the owner's key

Return value: Returns no values.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract

6.6 Buy Order Cancellation

Cancel the order to buy tokens.

```
void cancelBuyOrder(cell args_cl)
```

Method parameters encoded in cell args_cl:

1. price - A token sale price
2. min_amount - A minimum number of tokens the user is willing to trade
3. deals_limit -A maximum number of transactions for an instrument that can be matched in one cycle of the Order Execution Machine
4. value - An amount of TONs to deploy the contract
5. price_code - Price.tvc smart contract code in the Base64 encoding
6. tip3_code - TIP3 Wallet smart contract code in the Base64 encoding
7. ref<Tip3Config> tip3cfg - a reference to the structure of TIP3 Root configuration (name, symbol, decimals, root_public_key, root_address)

Limitations:

1. Method call is possible only by external message
2. The message must be signed with the owner's key

Return value: Returns no values.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract

6.7 Exchange Order Cancellation

Cancel the order for the exchange of tokens.

```
void cancelXchgOrder(cell args_cl)
```

Method parameters encoded in cell args_cl:

1. sell - false for sell, true for buy – direction of an order
2. price_num - a nominator of exchange price
3. price_denum - a denominator of exchange price
4. min_amount - A minimum number of tokens the user is willing to trade
5. deals_limit - A maximum number of transactions for an instrument that can be matched in one cycle of the Order Execution Machine
6. value - An amount of TONs to deploy the contract
7. xchg_price_code - Price.tvc smart contract code in the Base64 encoding
8. tip3_code - TIP3 Wallet smart contract code in the Base64 encoding
9. ref<Tip3Config> tip3cfg - a reference to the structure of TIP3 Root configuration (name, symbol, decimals, root_public_key, root_address)

Limitations:

1. Method call is possible only by external message
2. The message must be signed with the owner's key

Return value: Returns no values.

Possible errors:

- message_sender_is_not_owner the message is not signed by the owner of the contract

6.8 Coins Transfer

It is allowed to transfer the leftover coins from the contract balance to the selected address.

```
void transfer(address dest, uint128 value, bool_t bounce)
```

Method parameters encoded in cell args_cl:

1. dest - A destination address
2. value - A amount of nanoTONs to transfer
3. bounce - should the bounce message be generated

Limitations:

1. Method call is possible only by external message
2. The message must be signed with the owner's key

Return value: Returns no values.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract

6.9 Token Exchange Order

Place an order for the exchange of tokens with specified parameters

```
void deployPriceXchg(cell args_cl)
```

Method parameters encoded in cell args_cl:

1. sell - false for sell, true for buy – direction of an order
2. price_num - nominator of exchange price
3. price_denum - denominator of exchange price
4. min_amount - minimum number of tokens the user is willing to trade

5. deals_limit - maximum number of transactions for an instrument that can be matched in one cycle of the Order Execution Machine
6. value - An amount of TONs to deploy the contract
7. xchg_price_code - Price.tvc smart contract code in the Base64 encoding
8. tip3_code - TIP3 Wallet smart contract code in the Base64 encoding
9. ref<Tip3Config> tip3cfg - a reference to the structure of TIP3 Root configuration (name, symbol, decimals, root_public_key, root_address)

Limitations:

1. Method call is possible only by external message
2. The message must be signed with the owner's key

Return value: Returns no values.

Possible errors:

- message_sender_is_not_owner the message is not signed by the owner of the contract
- not_enough_balance - an error when land_amount is more than wallet balance
- not_enough_tons_to_process - an error when the sum of the total cost for operation and tokens cost in TON is more than TON balance
- unverified_tip3_wallet - in case pubkey or/and internal owner address doesn't own this wallet
- not_enough_tokens_amount - an error when an amount of tokens is less than permitted amount of tokens for one deal
- too_big_tokens_amount - in case amount then wallet balance of tokens or amount*price > 128 bit

6.10 Get Contract Owner Address

Place an order for the exchange of tokens with specified parameters.

```
address getOwner()
```

Limitations: No limitations.
Return value: Returns the FlexClient contract owner adress.

6.11 Get Flex Contract Address

The Flex contract contains important settings for the execution of transactions, such as the amount of commissions for transactions.

```
address getFleX()
```

Limitations: No limitations.
Return value: Returns the FlexClient contract owner adress.

7 RootTokenContract Smart Contract. API Description

The FlexToken token is architecturally similar to the TIP3 token. There is a root smart contract RootTokenContract, which is also involved in issuing tokens and transferring tokens to users' wallets.

User wallets are implemented as a smart contract TONTokenWallet. Wallets can accept tokens from RootTokenContract and other similar wallets. At the acceptance stage, it is checked that the tokens came from the user wallet of the same token as the receiving one.

7.1 Root Contract Deployment

```
void constructor(bytes name,
                 bytes symbol,
                 uint8 decimals,
                 uint256 root_public_key,
                 uint256 root_owner,
                 TokensType total_supply)
```

Constructor parameters:

1. name - A full name of the issued token
2. symbol - A symbolic name of the issued token
3. decimals - A number of decimals positions
4. root_public_key - A public key of the contract owner
5. root_owner - An address of the contract owner without workchain index
6. total_supply - A total amount of tokens issued into circulation (max 128 bit)

Limitations:

1. Method call is possible only by external message
2. The message must be signed with the owner's key

Return value: Returns no values.

Possible errors:

- define_pubkey_or_internal_owner - should be set owner public key or address only. Others should be set to 0

7.2 Wallet Creation

Creates a user wallet and transfers the specified number of tokens.

```
address deployWallet(int8 workchain_id,
                     uint256 pubkey,
                     uint256 internal_owner,
                     TokensType tokens,
                     WalletGramsType grams)
```

Parameters:

1. workchain_id - A workchain index
2. pubkey - A public key of the wallet owner
3. internal_owner - An address of the wallet owner without workchain index

4. tokens - An amount of tokens transfered to the wallet
5. grams - An amount of TONs to deploy the wallet

Limitations:

1. Method call is possible only by external message
2. The message must be signed with the owner's key

Return value: The address of the token wallet contract deployed to the blockchain.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract
- not_enough_balance - an error when the sum of granted Tokens and an amount of Tokens planning to grant to the wallet is more than total Tokens amount settled in the Constructor

7.3 Empty Wallet Creation

Creates an empty wallet for the user.

```
address deployEmptyWallet(int8 workchain_id,
                          uint256 pubkey,
                          uint256 internal_owner,
                          WalletGramsType grams)
```

Parameters:

1. workchain_id - A workchain index
2. pubkey - A public key of the wallet owner
3. internal_owner - An address of the wallet owner without workchain index
4. grams - An amount of TONs to deploy the wallet

Limitations:

1. Method call is possible only by external message

2. The message must be signed with the owner's key

Return value: The address of the token wallet contract deployed to the blockchain.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract

7.4 Tokens Transfer

The function transfers tokens to the destination token wallet.

```
void grant(address dest, TokensType tokens, WalletGramsType grams)
```

Parameters:

1. dest - A destination address of the token wallet
2. tokens - An amount of tokens transfered to the wallet
3. grams - An amount of TONs to deploy the wallet

Limitations:

1. The message must be signed with the owner's key

Return value: The address of the token wallet contract deployed to the blockchain.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract
- not_enough_balance - an error when the sum of granted Tokens and an amount of Tokens planning to grant to the wallet is more than total Tokens amount settled in the Constructor

7.5 Tokens Emission

Increases total amount of the tokens.

```
void mint(TokensType tokens)
```

Parameters:

1. tokens - An amount of tokens to increase the total number of tokens

Limitations:

1. The message must be signed with the owner's key

Return value: Returns no values.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract

7.6 Set Wallet Code

The function loads token wallet code for the future deploying of the Token owners wallets.

```
void setWalletCode(cell wallet_code)
```

Parameters:

1. wallet_code - The token wallet smart contract code in the Base64 encoding

Limitations:

1. The message must be signed with the owner's key

Return value: Returns no values.

Possible errors:

- message_sender_is_not_owner - the message is not signed by the owner of the contract

7.7 Get Token Information

It is possible to get various information about the token through functions with self-explanatory name.

```
string getName();
string getSymbol();
uint8 getDecimals();
uint256 getRootKey();
uint128 getTotalSupply();
uint128 getTotalGranted();
cell getWalletCode();
```

7.8 Get Wallet Address

Getting the wallet address based on the owner public key or address.

```
address getWalletAddress (int8 workchain_id, uint256 pubkey, uint256
    internal_owner)
```

Parameters:

1. workchain_id - A workchain index
2. pubkey - A public key of the wallet owner
3. internal_owner - An address of the wallet owner without workchain index

Limitations: No limitations.
Return value:
The calculated address of the token wallet contract.

Possible errors:
No errors.

7.9 Get Wallet Hashcode

Getting the wallet address based on owner public key or address.

```
uint256 getWalletCodeHash ()
```

Limitations: No limitations.

Return value:
The hashcode of the wallet code loaded into root contract.

Possible errors:
No errors.

8 TONTokenWallet Smart Contract. API Description

Here we provide a description of the API of custom wallets implemented as smart contract TONTokenWallet.

8.1 Tokens Transfer

Transfer Tokens from balance or granted balance by internal or granted owner.

```
void transfer (address dest,
               TokensType tokens,
               bool_t return_ownership,
               address answer_addr)
```

Parameters:

1. dest - A destination address of the token wallet
2. tokens - An amount of tokens to transfer
3. return_ownership - A flag to revoke the granted ownership
4. answer_addr - An address to send an answer message

Limitations:
This function can be called by current (internal or granted) owner only.

Return value:
No return value

Possible errors:

- internal_owner_disabled - an error when the internal owner address is set to 0 for the internal messages

- message_sender_is_not_my_owner - an error when the caller is not current owner (internal or granted)
- not_enough_balance - an error when the token balance is less than an amount to transfer or destination address is 0
- not_enough_tons_to_process - an error when the contract balance is less than an amount needed to process

8.2 Get Wallet Balance

Get the wallet token balance. This function can be called by internal owner or granted owner only. A result message will be sent to the caller contract function.

```
TokensType getBalance_InternalOwner()
```

Return value:
In case lend not granted, Tokens amount on the wallet. Otherwise, the minimum of the Tokens wallet balance or the lended balance
Possible Errors:

- internal_owner_disabled - an error when the internal owner address is set to 0 for the internal messages
- message_sender_is_not_my_owner - an error when the caller is not current owner (internal or granted)

8.3 Acceptance of Tokens

Accepts tokens from the root TIP3 contract only. This function can be called by root token contract only.

```
void accept(TokensType tokens)
```

Parameters:

1. tokens - An amount of tokens to accept from the root TIP3 contract

Limitations:
This function can be called by current (internal or granted) owner only.

Return value:
No return value.

Possible errors:

- message_sender_is_not_my_owner - an error when the caller is not current owner (internal or granted)

8.4 Internal Tokens Transfer

Transfers the tokens that are available only through sending a message from another smart contract.

```
void internalTransfer(TokensType tokens,
                      uint256 pubkey,
                      uint256 my_owner_addr,
                      address answer_addr)
```

Parameters:

1. tokens - An amount of tokens to transfer
2. pubkey - A public key of the wallet owner
3. my_owner_addr - An owner address
4. answer_addr - An address to send an answer message

Limitations:
This function can be called by current (internal or granted) owner only.

Return value:
No return value.

Possible errors:

- message_sender_is_not_good_wallet - an error when the sender is not TIP3 wallet

8.5 Deleting a Wallet

Deleting an empty wallet.

```
void destroy(address dest)
```

Parameters:

1. dest - An address to send the remaining coins from the wallet balance

Limitations:
This function can be called by the original internal owner only.
The wallet must not contain any tokens to proceed.

Return value:
No return value.

Possible errors:

- message_sender_is_not_good_wallet - an error when the sender is not the original wallet owner
- internal_owner_disabled - an error when the internal owner address is set to 0 for the internal messages
- destroy_non_empty_wallet - an error when balance of TIP3 tokens is non-zero
- only_original_owner_allowed - an error when the granted owner tries to destroy the wallet

8.6 Wallet Lending

Grants limited ownership of the wallet to the other user.

```
bool_t lendOwnership(uint256 std_dest,
                     TokensType lend_balance,
                     uint32 lend_finish_time,
                     cell deploy_init_cl,
                     cell payload)
```

Parameters:

1. std_dest - An address hashcode to grant permission to Price contract

2. lend_balance - An amount of tokens transferred ownership
3. lend_finish_time - Unix time of the end of transferred ownership
4. deploy_init_cl - A structure of deploy parametrs packed in cells to make an order
5. payload - A structure of the order parameters packed in cells for the order

Return value:
True.

Possible errors:

- not_enough_balance - an error when the token balance is less than amount to transfer ownership

8.7 Revoking Wallet Lending

Revokes lended ownership.

```
void returnOwnership()
```

Possible errors:

- internal_owner_disabled - an error when the internal owner address is set to 0 for the internal messages
- message_sender_is_not_my_owner - an error when the caller is not current owner (internal or granted)

8.8 Get wallet information

Get detailed information about the wallet.

```
details_info getDetails()
```

Return value:

- name - The name of the token

- symbol - The symbolic name (abbreviation) of the token
- decimals - The number of decimals positions of the token
- balance - The tokens balance on the wallet
- root_public_key - The public key of the TIP3 root contract owner
- root_address - The address of the token root contract
- owner_address - The address of the owner of the wallet contract
- lend_ownership structure:
    1. owner - The address of the lend ownership owner of the tokens
    2. lend_balance - The amount of tokens granted into lend ownership
    3. lend_finish_time - Unix time of the end of transfered ownership
- code - TIP3 Token wallet code
- allowance structure
    1. spender - The address of the lend ownership owner of the Tokens
    2. remainingTokens - The amount of tokens granted into lend ownership
- workchain_id - The workchain index