Formal verification for Molecula project

Version 1.1

Prepared by Pruvendo at 06/26/24

Executive summary

Pruvendo performed a formal verification (as an advanced form of an audit that dramatically decreases chance for important bugs) for smart contracts of Molecula project (https://github.com/molecula-fi/molecula-contracts, commit 505a5d93aa4429792febb98b480ddc36247695ea, made on 06/20/24), as well a traditional audit for web application of Molecula project (https://github.com/molecula-fi/molecula-monorepo, commit d0486bf7b721842fbcfff5908b5b690043ff17ec , made on 06/018/24).

While a number of bugs were found, all the major and critical ones were fixed by the development team. As a result Pruvendo gives a **positive** verdict and recommends the project to be released.

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.

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 development (it's based on a multistep 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 right job*, 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 terms, 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 collision 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 their 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
Parentheses	Combine the different *rectangles* into the same *outcome*
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:

- *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 consists of six types of 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 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 some 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`

Important!!! 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 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

This table 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 $\forall$ to `forall`, while Hoare logic is

[1] Currently this process is manual, the partial or full automation is planned in foreseeable future

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
- 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.

[2] Automation of *Stage7* is in the roadmap

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 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**.

Description of the audit approach

While only smart contracts are supposed to be verified, the web application is just audited. The audit is performed manually, by at least two engineers and follows CWE-699 standard. In particular, the following potential weakness are checked:

- Access control errors
- Bad coding practices, such as:
    - Misleading identifiers
    - Code difficult to be understood
    - Lack of decomposition
    - Too long functions
    - Spaghetti code
    - Mixing of different abstraction layers
    - Too many layers of inner blocks
    - Unneeded asynchronous code
    - Other bad code practices
- Behavioural errors
- Business logic errors
- Communication channel errors
- Complexity issues
- Concurrency issues
- Credential management errors
- Data integrity issues
- Data processing errors
- File handling issues
- Encapsulation issues
- Error conditions, return values, status codes
- Expression issues
- Initialization and cleanup errors
- Validation issues
- Numeric errors
- Random number issues
- String errors
- Performance issues

CWE-699 defines some other types of error, but they are considered as redundant for this particular project. All the bugs and notes are sent to the development team and, after discussion, one of the following decisions is applied for each of them:

- Rejected as invalid
- Accepted and not fixed, as minor and insignificant
- Accepted and marked as to be fixed

The positive outcome of the audit is not possible until all the “to be fixed” bugs are actually fixed. The audit process is iterative, starting from roughly “feature freeze” of the project and repeated a few times until the final release is made.

Formal verification

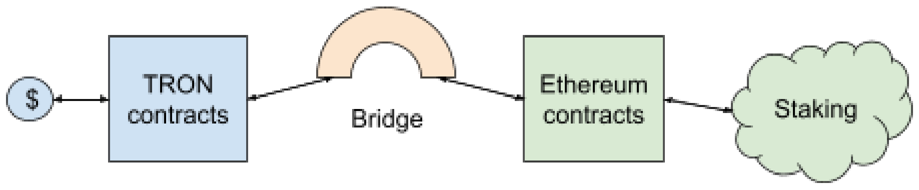
Business-level specification

Purpose

The purpose of the present sectionis to specify the Molecula project 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 Molecula.

Introduction

Currently TRON network experiences lack of investments products that prevents many holders from profit generation. At the same time Ethereum network provides a whole bunch of effective staking products. The idea of the Molecula project is to let TRON users to transfer their assets into Ethereum, stake them, get profit and, finally, withdraw their stakes in the original form of TRON assets, as demonstrated at the illustration below.



So, the user provides his TRON assets to the “TRON contracts”, their equivalent is transferred through the bridge to “Ethereum contracts”, where staked. Upon withdrawal, the process is reversed. Some fee is applicable.

The following solutions are selected for each component mentioned above:

- TRON assets - TRC20 USDT[3]
- TRON contracts - to be developed as a part of Molecula project (in Solidity)
- Bridge

[3] Here and later USDT stays for any USD stable coin finally selected for the project (USDT, USDC, DAI, etc.)

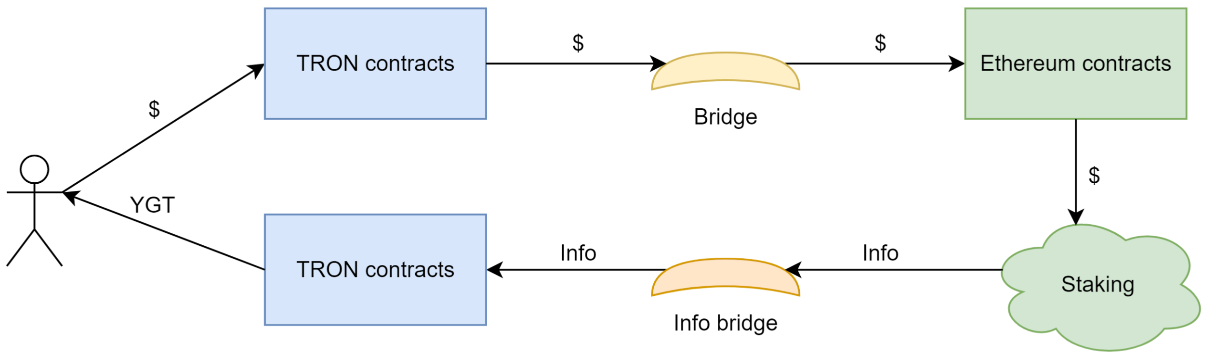
    - For transferring of assets - Swft
    - For internal communication - to be developed as a part of Molecula project (in Typescript) (later it's called **info bridge**)
- Ethereum contracts - to be developed as a part of Molecula project (in Solidity)
- Staking - Spark

As the present business-level specification is solely intended to be a basement for a formal specification to be developed, only "TRON contracts" and "Ethereum contracts" are considered throughout the present document as only these components are subject for the formal verification.

YGT Tokens

YGT Tokens are TRC20 tokens that are granted to the user in exchange for his deposit. Basically, when depositing:

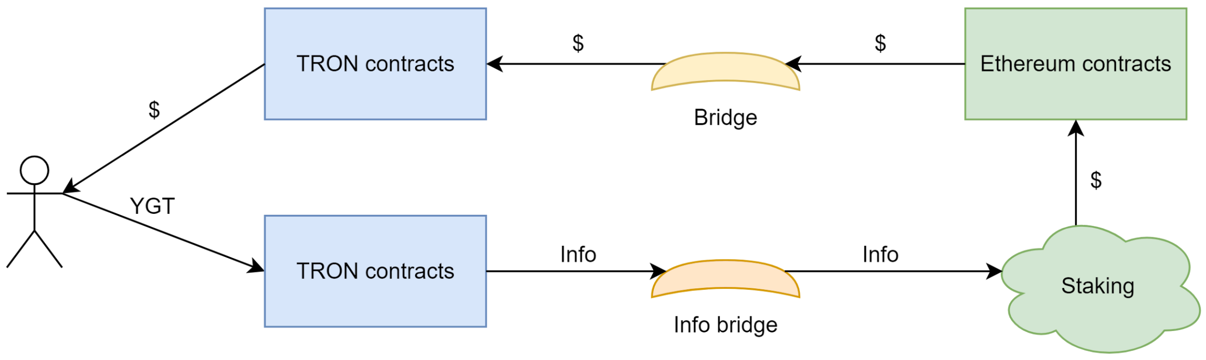
- The user provides his TRON USDT tokens to the Molecula
- USDT TRC20 tokens are converted into USDT ERC20 tokens via bridge
- USDT ERC20 tokens are staked
- The whole idea of YGT tokens is that so called "shares" are kept rather than tokens themself. A number of tokens (*balanceOf*) is dynamic and calculated according to the following formula: $b = \frac{ps}{t}$, where *b* - balance of tokens, *s* - number of shares owned by the user, *t* - total number of shares, p - total amount of USDT deposited in the pool
- Amount of YGT tokens to be minted equals to the amount of USDT deposited, excluding the Molecula service fee and the bridge fee
- Molecula mints the corresponding amount of YGT TRC20 shares and transfers them to the user
- Graphically it looks as below



Upon withdrawal :

- Amount of USDT to withdraw is calculated by the following formula: $a = \frac{sp}{t}$, where *s* - number of YGT shares to be sold, *t* - total number of YGT shares, *p* - pool size in USDT
- The YGT shares to be sold are burnt
- The required amount is withdrawn from staking

- USDT ERC20 tokens are converted into USDT TRC20 tokens via bridge
- USDT TRC20 tokens are returned to the user



More than one YGT can be supported.

YGT rate

At any time YGT share rate is calculated according to the following formula: $r = \frac{p}{t}$, where *r* - the current rate, *p* - amount of USDT staked, *t* - total amount of YGT shares. The rate of YGT token is always 1 USDT.

Detailed steps

As it was mentioned above, business logic located in smart contracts is discussed here exclusively. The rest of business logic is subject to audits, not to the formal verification.

Deposit

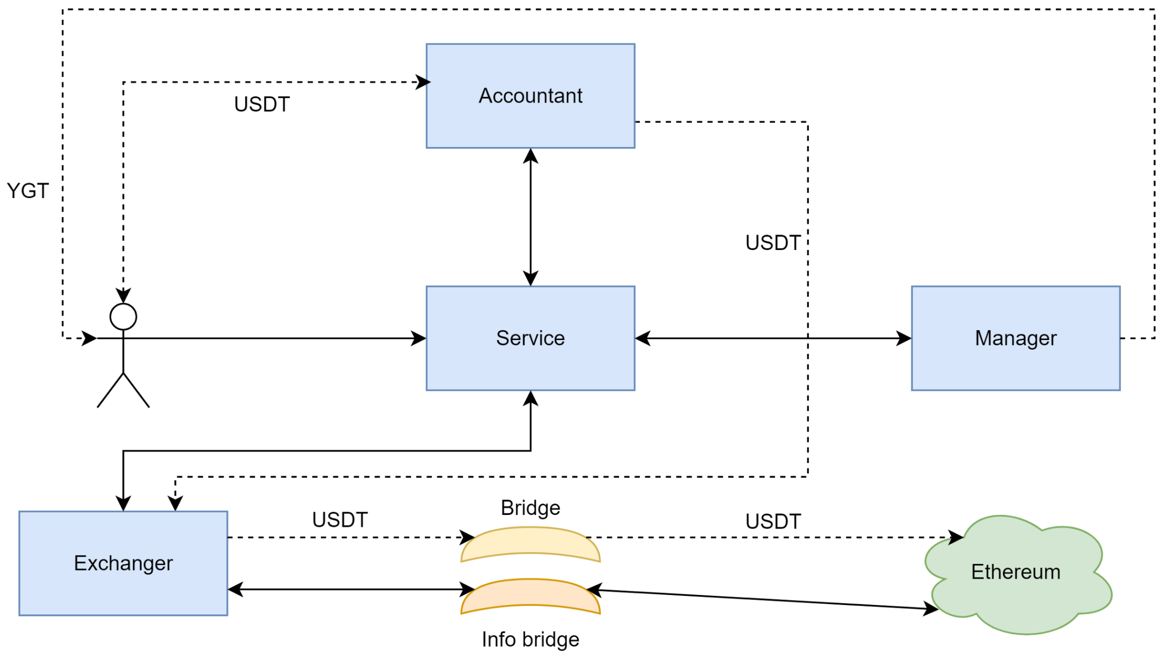
TRON part

It's assumed that before starting the process, the user already allowed the required amount of USDT to the entity called *Accountant*[4].

1. The user initiates the operation providing the following information and sending a request to the entity called *Service*:
    a. Deposit amount
    b. Selected YGT token
2. *Service* :
    a. Checks if YGT token exists
    b. Generates operation *UUID*
    c. Sends the information about the operation (including *UUID*) to the entity called *Manager*
3. *Manager*:
    a. Checks if:

[4] Here and later any entity can be either a separate smart contract or just an abstract virtual construction

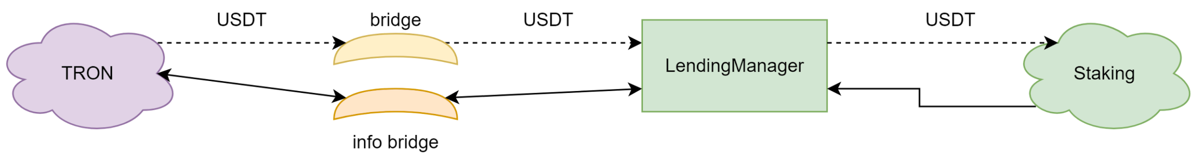
        i. The required amount is allowed and available
        ii. The required amount is positive and exceeds minimally allowed value
    b. Stores the information provided by the *Service*
4. *Service* send the information about the operation to the *Accountant*
5. *Accountant*:
    a. Obtains the USDT from the user
    b. Calculates fee (to be described separately)
    c. Transfers the whole amount subtracted by fee to the entity called *Exchanger*
    d. Locks the fee until the moment confirmation is received
6. *Service* informs the *Exchanger* about the operation
7. *Exchanger* sends the USDT to the bridge
8. Positive operation result is received from the info bridge (or not received) to the *Exchanger*
    a. *Exchanger* sends the operation result to the *Service*
    b. *Service* sends the operation result to the *Manager*
    c. *Manager*:
        i. Mints the required amount of YTG shares and assigns them to the user
        ii. Puts the operation into the “completed” status
        iii. Sends the operation back to the *Service*
    d. *Service* sends the operation result to the *Accountant*
    e. Accountant unlocks the fee
9. Negative operation result is received from the info bridge to the *Exchanger*: In this case:
    a. *Exchanger* sends the operation result to the *Service*
    b. *Service* sends the operation result to the *Manager*
    c. *Manager* marks the operation as “reverted” and returns it to the *Service*
    d. *Service* sends the operation result to the *Accountant*
    e. *Accountant* returns all the assets back to the initiator
10. Graphically, the relations between the components can be illustrated by the diagram below (here and below solid lines stay for control pass (messages, subroutines etc.), while dotted lines - for simple transfer of assets)



Ethereum part

The only entity from the Ethereum part is *LendingManager*. Assuming it received the deposit from the bridge in USDT, the next actions are as follows:

1. The lending process is initiated by the info bridge
2. *LendingManager*:
    a. Sends the assets to the Staking
    b. Waits (or synchronously gets) for the result of arranging the deposits
    c. Calculates amount of YGT token shares to be minted as well as an updated total amount of YGT token shares
    d. Sends the result to the info bridge
3. Graphically, the relations can be illustrated by the following diagram



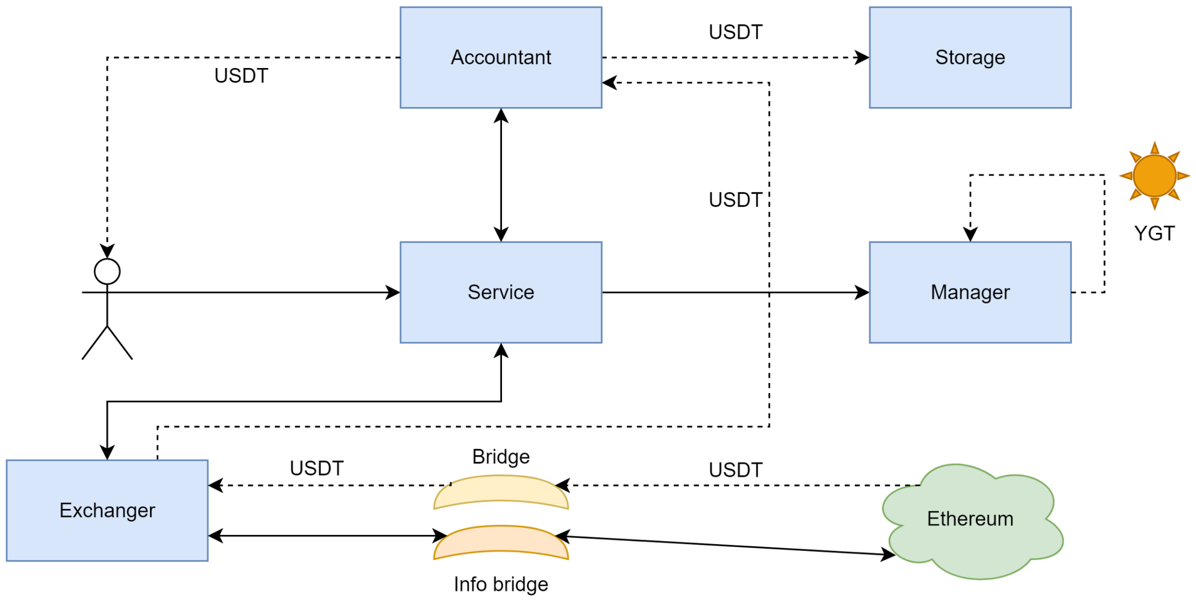
Withdrawal

TRON part

Unlike for the “deposit” operation there is no need to allow YGT tokens to anybody, as the *Manager* has allowance for all the operations with YGT tokens.

1. The user initiates the operation providing the following information and sending a request to the *Service*:

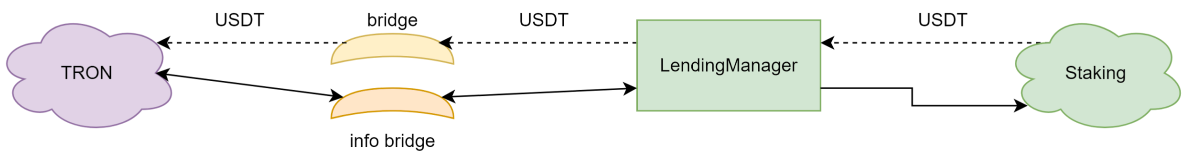
    a. Withdrawal amount
    b. Selected YGT token
2. *Service* :
    a. Checks if YGT token exists
    b. Generates operation *UUID*
    c. Sends the information about the operation (including *UUID*) to the *Manager*
3. *Manager*:
    a. Checks if the required amount is owned by the user
    b. Stores the information provided by the *Service*
    c. Locks the user's YGTs
    d. Returns the information to the Service
4. *Service* informs the *Exchanger* about the operation
5. *Exchanger* sends the information about the operation to the info bridge
6. Info bridge sends the message to the Exchanger when the latter receives the assets (USDT) via bridge
7. Positive operation result is received from the info bridge (or not received) by the *Exchanger*
    a. *Exchanger* provides a required allowance to the *Accountant*
    b. Operation result is sent to the *Service*
    c. *Service* sends the operation result to the *Manager*
    d. *Manager*:
        i. Burns the locked YGT tokens
        ii. Marks the operation as "completed"
        iii. Returns the operation information back to the *Service*
    e. *Service* informs the *Accountant* about the operation
    f. *Accountant*:
        i. Takes the required amount from the *Exchanger*
        ii. Send all the amount, but fee, to the initiator
8. Negative operation result is received from the info bridge (or not received) by the *Exchanger*
    a. Operation result is sent to the *Service*
    b. *Service* send the operation result to the *Manager*
    c. Manager unlocks the YGT tokens by assigning them back to the initiator
9. Graphically, the relations between the components can be illustrated by the diagram below



Ethereum part

As above the *LendingManager* is the only entity to be considered.

1. The lending process is initiated by the info bridge
2. *LendingManager* sends the request for withdrawal to the staking
3. Upon retrieval the assets the *LendingManager*:
    a. Sends the information to the info bridge
4. the info bridge sends the request to the *LendingManager* with bridge request details
5. LendingManager:
    a. Sends the request to the bridge
    b. Sends the result to the info bridge
6. Graphically, the relations can be illustrated by the following diagram



Rate acquiring

As it was mentioned above, the YTG rate is calculated by the formula $\frac{p}{t}$, where *p* - the total amount of the staked assets, *t* - the total amount of YTG tokens. It's important to mention that both numbers are handled at the Ethereum side.

Thus:

- At the TRON side there is an entity called *Oracle* that:
    - Provides the current rate by request
    - Total amount of the staked assets is regularly updated by the info bridge

- At the Ethereum side the info bridge regularly updates the rate by getting the values from:
    - Staking - to get the pool amount

High-level specification

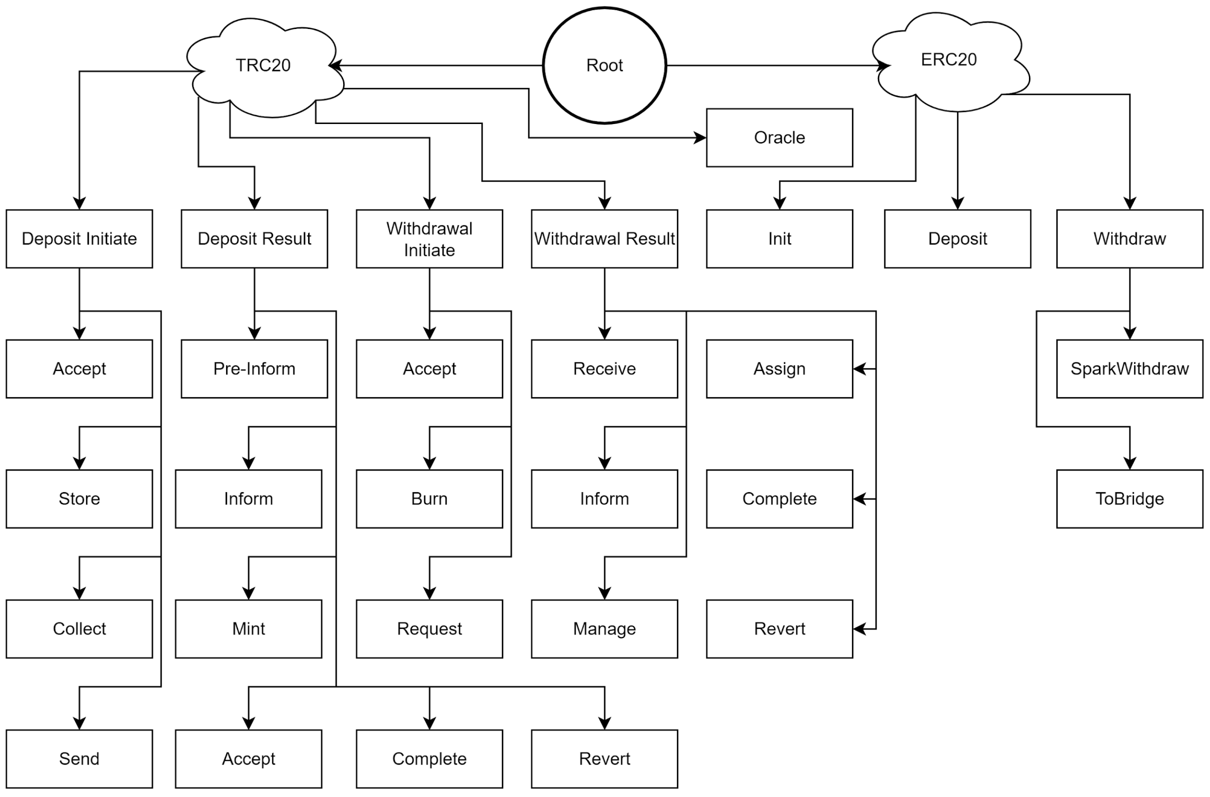
As it was mentioned above, the high-level specification is based on the following principles:

- 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 root user scenarios were identified:

- TRC20 part
    - Oracle Scenario - changing number of pool and shares to keep the liquid rate of YTG tokens correct
    - Deposit Initiate Scenario - sending the USDT into bridge to convert them to ERC20
    - Deposit Result Scenario - receiving an answer from the bridge about deposit success/failure, generation of YTG tokens, sending them to the initiator, fee collection
    - Withdrawal Initiate Scenario - receiving YGT tokens from the initiator, sending the request to the bridge for the USDT
    - Withdrawal Result Scenario - receiving an answer from the bridge about deposit success/failure, burning the YTG tokens, sending USDT to the initiator, fee collection
- ERC20 part
    - Init Scenario - Initialization of the ERC20 system
    - Deposit Scenario - receiving USDT from the bridge, depositing them in the stake pool, receiving the *spUSDT* tokens and sending them to the info bridge
    - Withdrawal Scenario - receiving the *spUSDT* tokens from the info bridge, returning them to the stake pool, getting USDT from the stake pool and sending them to the bridge

During the later decomposition the following tree of subscenarios has been 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/27ed473e-6397-4461-b3f3-3abdc16d9a16 .

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
  - The secondary workflow is performed via ⇗ and ⇘ buttons
- 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

ERC20 scenarios

Axioms

ID	Human description	Formal description
		$\forall\sigma \in SparkPool,\ u = 'USDC',\ \omega = 'SPUSDC',\ k \in \aleph,\ b = balanceOf,\ a = allowance$
		$\varrho \in ISwftSwap, y \in YGTExchanger, \tau \in String,\ \tau_0 = 'USDT(TRON), Y \subset YGTEchanger'$
EI.1	Balance of 'USDC' token is decreased accordingly when deposited	$\{\forall x \in \aleph: x = u.b(this), y = u.a(this,\sigma): x \geq k \wedge y \geq k\}$ $\sigma.supply(u, k, this, 0)$ $\{u.b(this) = x - k \wedge u.a(this,\sigma) = y - k\}$
EI.2	Balance of 'SPUSDC' token is increased accordingly when deposited	$\{\forall x \in \aleph: x = \omega.b(this),\ u.b(this) \geq k \wedge u.a(this,\sigma) \geq k\}$ $\sigma.supply(u, k, this, 0)$ $\{\omega.b(this) = x + k\}$
EI.3	In case of insufficient balance or allowance an attempt to deposit fails	$\{\ u.b(this) < k \vee u.a(this, \sigma < k)\}$ $\sigma.supply(u, k, this, 0)$ $\{S = false\}$
EW.1	In case of withdrawal the returned amount does not exceed the requested one	$\{\forall x, y: x = u.b(this) \wedge\ y = \omega.b(this) \wedge k \leq y \wedge\}\kappa = \sigma.withdraw(u, k, this)$ $\{u.b(this) = x + \kappa \wedge \omega.b(this) = y - \kappa \wedge \kappa \leq k\}$ $\{\forall x, y: \neg(x = u.b(this) \wedge\ y = \omega.b(this) \wedge k \leq y)\}\sigma.withdraw(u, k, this)\{S = false\}$

ER.1	In case of transfer and enough balance the sender's balance it decreased by the required amount while receiver's balance is increased by the same amount	$\{\forall x, y \in A,\ b_x, b_y, k \in \aleph,\ t \in IERC20: t.b(x) = b_x \land t.b(y) = b_y \land b_x \geq k \land msg.sender = x \land$ $x \neq y\} t.transfer(y, k)$ $\{t.b(x) = b_x - k \land t.b(y) = b_y + k\}$
ER.1a	In case of transfer and insufficient balance, zero address of the sender or zero address of the recipient, exception is thrown	$\{\forall x, y \in A,\ b_x, b_y, k \in \aleph,\ t \in IERC20: t.b(x) = b_x \land t.b(y) = b_y \land msg.sender = x \land$ $\land \left(b_x < k \lor x = 0 \lor y = 0\right)\} t.transfer(y, k)$ $\{S = false\}$
ER.1b	In case the sender and the recipient are the same nothing happens	$\{\forall x, y \in A,\ b_x, b_y, k \in \aleph,\ t \in IERC20: t.b(x) = b_x \land t.b(y) = b_y \land b_x \geq k \land msg.sender = x \land$ $x = y\} t.transfer(y, k)$ $\{t.b(x) = b_x \land t.b(y) = b_y\}$
ER.2	In case of approve the allowance becomes equal to the required one	$\{\forall x, y \in A, t \in IERC20,\ k \in \aleph: x = msg.sender \land x \neq 0 \land y \neq 0\} t.approve(y, k)$ $\{t.a(x, y) = k\}$
ER.2a	If an approver or an approvee is zero, the exception is raised	$\{\forall x, y \in A, t \in IERC20,\ k \in \aleph: x = msg.sender \land (x = 0 \lor y = 0)\} t.approve(y, k)$ $\{S = false\}$
ER.3	In case of transferFrom, enough balance and allowance, balances and allowances are changed accordingly	$\{\forall x, y, z \in A,\ b_x, b_z, a, k \in \aleph,\ t \in IERC20: t.b(x) = b_x \land t.b(z) = b_z \land b_x \geq k \land$ $t.a(x, y) = a \land a \geq k \land x \neq z \land y = msg.sender \land$ $x > 0 \land z > 0\}$ $t.transferFrom(x, z, k)$ $\{t.b(x) = b_x - k \land t.b(z) = b_z + k \land t.a(x, y) = a - k\}$

ER.3a	If balance is insufficient, sender is zero or recipient is zero, transferFrom must fail	$\{\forall x, y, z \in A,\ b_x, b_z, a, k \in \aleph,\ t \in IERC20: t.b(x) = b_x \land t.b(z) = b_z \land t.a(x, y) = a \land a \geq k \land y = msg.sender \land \left(b_x < k \lor x = 0 \lor z = 0\right)\}$ $t.transferFrom(x, z, k)$ $\{S = false\}$
ER.3b	If sender and recipient are the same, while all other requirements are met, nothing happens, but allowance decrease	$\{\forall x, y, z \in A,\ b_x, b_z, a, k \in \aleph,\ t \in IERC20: t.b(x) = b_x \land t.b(z) = b_z \land b_x \geq k \land t.a(x, y) = a \land a \geq k \land x = z \land y = msg.sender \land x > 0 \land z > 0\}$ $t.transferFrom(x, z, k)$ $\{t.b(x) = b_x \land t.b(z) = b_z \land t.a(x, y) = a - k\}$
EM.1	`MesonBridge.tokenForIndex` returns a token address if registered or fails. Address for USDC is returned if and only if the index was correct	$\forall i: i = I \Leftrightarrow MesonBridge.tokenForIndex(i) = u$
EM.2	`MesonBridge.postSwapFromContract` is successful if and only if: ● The sender has enough balance and allowance ● `isAuthorized` callback returns true ● The currency is correct ● Some additional requirements In this case the balance and	$\{\forall b, b_m, a, k, s \in \aleph, c, i \in A, m \in IMesonBridge: b = u.b(c) \land a = u.a(c, m) \land b \geq k \land a \geq k \land (s >> 208)\ \&\ 0xFFFFFFFFFF = k \land b_m = u.b(m) \land (s\ \&\ 0xFFFFFFFF) = I \land c = msg.sender\}$ $m.postSwapFromContract(s, 2^{40}i + 1, c)$ $\left\{u.b(c) = b - k \land u.a(c, m) = b - k \land u.b(m) = b_m + k\right\}$ $\{\forall b, a, k, s \in \aleph, c, i \in A, m \in IMesonBridge: \neg(b = u.b(c) \land a = u.a(c, m) \land b \geq k \land a \geq k \land (s >> 208)\ \&\ 0xFFFFFFFFFF = k \land (s\ \&\ 0xFFFFFFFF) = I \land c = msg.sender)\}$ $m.postSwapFromContract(s, 2^{40}i + 1, c)$

	allowance are decreased and increased accordingly	$\{S = false\}$
ES.1	The `SwftBridge` swap is correct if and only if: ● Balance exceeds the required amount ● Allowance exceeds the required amount ● `usdcTokenAddress` corresponds to 'USDC' ● `fromAmount` is 'USDC(TRON)' ● `destination` is a known YTG	$\forall b, b_s, a, k, m \in \aleph, \upsilon \in A, \phi \in String, \varrho \in SwftBrifge:$ $\{b = u.b(c) \land b_s = u.b(\varrho) \land a = u.a(c, \varrho) \land k \geq b \land k \geq a \land \upsilon = u \land y \in Y \land \tau = \tau_0\}$ $\varrho.swap(\upsilon, \tau, y, k, m)$ $\{u.b(c) = b - k \land u.b(\varrho) = b_s + k \land u.a(c, \varrho) = a - k\}$ $\{\neg(b = u.b(c) \land b_s = u.b(\varrho) \land a = u.a(c, \varrho) \land k \geq b \land k \geq a \land \upsilon = u \land y \in Y \land \tau = \tau_0)\}$ $\varrho.swap(\upsilon, \tau, y, k, m)$ $\{S = false\}$

Scenario ERC20 Init

Mapping

Original element	Mapped element	Comments
σ	`sparkPoolAddress`	
b	`swftBridgeAddress`	
d	`initialStakeValue`	

s	`msg.sender`	
s_1	`msg.sender`	Equal to s_1 value
s_2	`msg.sender`	Equal to s_2 value
$l.m$	`LendingManager.initialStake`	
$l.b$	`LendingManager.swftBridge`	
$l.s$	`LendingManager.sparkPool`	
$l.t$	`LendingManager.totalSharesSupply`	
$l.o$	`LendingManager.owner`	
i	`LendingManager.constructor`	

Low-level invariants

N	Human description	Formal description
		$\forall l \in LendingManager,\ m, u, s, s_1, s_2 \omega, o \in A, d \in \aleph: u = 'USDC',\ \omega_0 = 'SPUSDC'$
EII.1	Anybody can initialize	$\dfrac{\{\forall b \in B: s_1 = msg.sender\} l.constructor(m,u,s,\omega,o,d)\{S=b\}}{\{\forall b \in B: s_2 = msg.sender\} l.constructor(m,u,s,\omega,o,d)\{S=b\}}$
EII.2	Initialization is successful if and only if: ● Balance is more than	$\{\forall b: B: b = (u.b(this) \geq d \wedge\ d\ > 0) \wedge$...

	required ● Required deposit is more than minimal ●	
EII.3	Ledger is not initialized beforehand	
EIO.1	Ledger is initialized afterward	$\{\ \}l.constructor(m, u, s, \omega, o, d)\{\curvearrowright l.constructor\}$
EIO.2	Bridge is assigned	$\{\ \}l.constructor(m, u, s, \omega, o, d)\{l.safeVault = m\}$
EIO.3	Spark is assigned	$\{\ \}l.constructor(m, u, s, \omega, o, d)\{l.sparkPool = s\}$
EIO.4	Total amount is the initial deposit	$\{\ \}l.constructor(m, u, s, \omega, o, d)\{l.totalSharesSupply = d\}$
EIO.5	Spark tokens arranged to the LendingManager equal to the deposit	$\{\forall x: x = \omega.b(this)\}l.constructor(m, u, s, \omega, o, d)\{l.spusdcToken = \omega \wedge \omega.b(this) = d + x\}$
EIO.6	Owner is assigned	$\{\ \}l.constructor(m, u, s, \omega, o, d)\{l.\_owner = d\}$
EIO.7	Balance is decreased by initial deposit	$\{\forall x, y: x = u.b(this)\}l.constructor(m, u, s, \omega, o, d)\{u.b(this) = x - d\}$
EIO.8	Spark is deposited by the assigned amount	$\{\forall x: x = u.b(s)\}l.constructor(m, u, s, \omega, o, d)\{u.b(s) = x + d\}$
EIO.9	Nothing else is changed	Specified directly at Stage 7

Scenario ERC20 Deposit

Mapping

Original element	Mapped element	Comments
s	`msg.sender`	
v	`value`	
d	`LendingManager.deposit`	
l.o	`LendingManager._owner`	

Low-level invariants

N	Human description	Formal description
		$\forall l \in LendingManager,\ s \in A, v, i \in \aleph: u = 'USDC',\ \omega = 'SPUSDC',\ d = deposit$
		$b = balanceOf,\ a = allowance$
EDI.1	Only owner can deposit	$\{msg.sender \neq l.\_owner)\}d(i,v)\{S = false\}$
EDI.2	Deposit is successful if and only if: ● Deposit is initiated by the Owner ● Deposit exceeds	$\{\forall b \in B: b = (msg.sender = l.\_owner \land\ v \leq u.b(this) \land l.deposits[i] = \varnothing)\}$ $d(i,v)$ $\{S = b\}$

	minimal amount ● Owner's balance exceeds deposit	
		$\{\ \}d(i,v)\{S = true\} \Leftrightarrow$
EDO.1	Owner's balance is decreased by deposit	$\{\forall x: x = u.b(this)\}d(i,v)\{u.b(this) = x - v\}$
EDO.2	Spark's balance is increased by the deposit	
EDO.3	Amount of the deposit tokens owned by the Owner is increased by the deposit	$\{\forall x: x = \omega.b(this)\}d(i,v)\{\omega.b(this) = x + v\}$
EDO.4	Total deposit amount is increased by deposit amount	Same as EDO.3
EDO.5	Shares are increased proportionally to the deposit	$\{\forall x: x = l.totalSharesSupply\}d(i,v)\left\{l.totalSharesSupply = x + \frac{v \cdot l.totalSharesSupply}{\omega.b(this)}\right\}$
EDO.6	Nothing else is changed	Specified directly at Stage 7

Scenario ERC20 Withdrawal

Mapping

Original element	**Mapped element**	**Comments**

s	`msg.sender`	
o	`operationId`	
w	`withdraw`	
b	`transferToMeson`	

All the high-level statements here are dropped in favor of subscenarios!!!!

Scenario [ERC20 sparkWithdraw]()

Low-level spec

N	Human description	Formal description
		$\forall l \in LendingManager,\ s \in A, v, i, mv \in \aleph: u = 'USDC',\ \omega = 'SPUSDC',\ w = withdraw$
		$b = balanceOf,\ a = allowance$
ESI.1	Only 'ERC20 Withdrawal' can initiate this scenario	$\{msg.sender \neq l.\_owner\}w(i, v)\{S = false\}$
ESI.1a	Swft version	$\{msg.sender \neq l.\_owner\}w(i, v, mv)\{S = false\}$
ESI.2	Amount must be positive	$\{l.totalSharesSupply = 0\}w(i, v)\{S = false\}$
ESI.2a	Swft version	$\{l.totalSharesSupply = 0\}w(i, v, mv)\{S = false\}$

ESI.2b	If it's a known operation, the scenario must faile	$\{l.withdrawal \neq \emptyset\} w(i, v, mv) \{S = false\}$
ESI.3	Amount must not exceed balance minus reserved amount	$\{v > l.totalSharesSupply - l.initialStake\} w(i, v) \{S = false\}$
ESI.3a	Swft version	$\{v > ω.b(this) - l.initialStake\} w(i, v, mv) \{S = false\}$
ESI.3b	minValue should be not less than value (Swft only)	
ESI.4	Otherwise, operation succeeds (Meson only)	$\{msg.sender = l.\_owner \wedge v > 0 \wedge v \leq ω.s(this) - l.initialStake\} w(i, v) \{S = true\}$
ESI.4a	Otherwise, operation succeeds, if the full amount was returned (Swft only)	$\{\forall x \in \aleph: msg.sender = l.\_owner \wedge v > 0 \wedge v \leq ω.s(this) - l.initialStake \wedge l.safeVault \neq 0 \wedge x = \frac{v \cdot u.s(this)}{l.totalSharesSupply} \wedge l.withdrawals[u] = \emptyset\}$ $w(i, v), κ = s.withdraw(u, v, l) \{S = (κ = x \wedge mv \leq x)\}$
ESI.5	Operation is not created and not added to the list of operations beforehand	List of operations is external
		$\{\ \} κ = s.withdraw(u, v, l) \{S = true\} \Leftrightarrow$
ESO.1	USDT balance of the owner is increased accordingly	$\{\forall x: x = u.b(this)\} w(i, v) \{u.b(this) = x + κ\}$
ESO.1a	USDT balance of the owner is the same (combination with EBO.1) (Swft version only)	$\{\forall x: x = u.b(this)\} w(i, v) \{u.b(this) = x\}$

ESO.2	USDT balance of the Spark in decreased by 'x'	
ESO.3	SPUSDT balance of the Owner is decreased by 'x'	$\{\forall x: x = \omega.s(this)\}\kappa = w(i, v)\{\omega.s(this) = x - \kappa\}$
ESO.4	Withdrawn amount less or equal than required	$\{\forall x, y \in \aleph: x = l.totalSharesSupply, y = u.b(this)\}w(i, v, mv)\left\{\kappa \le \frac{vy}{x}\right\}$
ESO.5	Amount of shares is decreased accordingly	$\{\forall x: x = l.totalSharesSupply\}w(i, v, mv)\{l.totalSharesSupply = x - v\}$
ESO.6	Operation is created, added to the list and its amount equals to real withdrawn amount	
ESO.7	Nothing else is changed	Specified directly at Stage 7

Scenario ERC20 toBridge

Low-level spec

N	Human description	Formal description
		$\forall l \in LendingManager,\ s \in A, v, i, d \in \aleph: u = 'USDC',\ \omega = 'SPUSDC',\ t = transferToMeson$
		$\forall \mu \in \aleph, \varrho = l.transferToSwftBridge,\ w = l.withdraw:$
		$b = balanceOf,\ a = allowance,\ m = l.mesonBridge,\ v = (d \ >> \ 208)\ \&\ 0xFFFFFFFFFF$

EBI.1	Scenario can be initiated by 'ERC20 Withdrawal' scenario only (Meson)	$\{msg.sender \neq l.\_owner\}l.transferToMeson(d, i)\{S = false\}$
EBI.1a	Scenario can be initiated by 'ERC20 Withdrawal' scenario only (Swft)	Covered by ESI.1
EBI.2	Currency must be correct (Meson version)	$\{d\&0xFFFFFFFF \neq I\}l.transferToMeson(d, i)\{S = false\}$
EBI.2a	Swft version	Not actual
EBI.3	Amount must not exceed balance (Meson version)	$\{v > u.b(l)\}l.transferToMeson(d, i)\{S = false\}$
EBI.3a	Swft version	Covered by ESI.3
EBI.4	Otherwise, the scenario is successful	$\{msg.sender = l.owner \wedge d\&0xFFFFFFFF = I \wedge v \leq u.b(l)\}l.transferToMeson(d, i)\{S = true\}$
EBI.4a	Swft version	Covered by ESI.4
EBI.5	Operation is in 'operations'	
EBO.1	Owner's balance is decreased by amount (Meson)	$\dfrac{\{\ \}l.transferToMeson(d,i)\{S=true\}}{\{\forall b \in \aleph : b=u.b(l)\}l.transferToMeson(d,i)\{u.b(l)=b-v\}}$
	Swft version	Covered by ESO.1
EBO.2	Bridge balance is increased by amount	$\dfrac{\{\ \}l.transferToMeson(d,i)\{S=true\}}{\{\forall b \in \aleph : b=u.s(m)\}l.transferToMeson(d,i)\{u.s(m)=b+v\}}$

		$\frac{\{\ \}w(i,v,mv)\{S=true\}}{\{\forall b\in\aleph : b=u.s(m)\}w(i,v,mv)\{u.s(m)=b+v\}}$
EBO.3	Operation is removed from 'operations'	
EBO.4	Nothing else is changed	Specified directly at Stage 7

TRC20 Scenarios

Axioms

ID	Human description	Formal description
DS.1	In case all the parameters are correct the sending to Swft bridge is successful (no exception)	$\forall\sigma \in ISwftBridge,\ \$_t = 'USDT', \$_e = 'USDC',\ v, mv \in \aleph,\ l \in address, b \in boolean$ $\{b = (\$_t.balanceOf(msg.sender) \geq v \land \$_t.allowance(msg.sender,\ \sigma) \land v > 0 \land v \geq mv)\}$ $\sigma.swap(\$_t, \$_e, l, v, mv)\{S = b\}$

Scenario TRC20 Oracle

Mapping

Original element	Mapped element	Comments
s	`msg.sender`	

p(init)	`initialPool`	
t(init)	`initialShares`	
o(init)	`initialOwner`	
p(setTotalSupply)	`pool`	
t(setTotalSupply)	`shares`	
i	`Oracle.init`	
sa	`Oracle.setPoolAmounts`	
l.o	`YGTToken.oracle`	
l.o.t	`YGTToken.oracle._shares`	
l.o.p	`YGTToken.oracle._pool`	
l.o.o	`YGTToken.oracle._owner`	

Low-level spec

N	Human description	Formal description
		$\forall\, t, p \in \aleph,\ o = Oracle(YGTToken.oracle),\ io \in address,\ b \in boolean$
RTI.1	Anybody can create an oracle, both numbers must be positive	Waived

		$\{b = (p > 0) \wedge (t > 0) \wedge \neg\curvearrowright o.constructor\}o.constructor(t, p, io)\{S = b\}$
RTI.2	Only InfoBridge can update amounts, both numbers must be positive	$\{b = (msg.sender = o.owner\}$ $o.setTotalSupply(p, t)$ $\{S = b\}$
RTO.1	init leads to creating the pool with initial amount	$\frac{\{\ \}o.constructor(t,p,io)\{S=true\}}{\{\ \}o.constructor(t,p,io)\{\curvearrowright o.constructor \wedge o.pool=p \wedge o.shares=t \wedge o.owner=io\}}$
RTO.2	Nothing else is changed while init	Specified directly at Stage 7
RTO.3	setAmounts leads to updating the amounts	$\frac{\{\ \}o.setTotalSupply(p,t)\{S=true\}}{\{\ \}o.setTotalSupply(p,t)\{o.pool=p \wedge o.shares=t\}}$
RTO.4	Nothing else is changed while setTokenAmount	Specified directly at Stage 7

Scenario TRC20 Deposit Initiate

Mapping

Original element	**Mapped element**	**Comments**
s_1	`msg.sender`	Equal to s_1 value
s_2	`msg.sender`	Equal to s_2 value
y	`exchanger`	

a	`value`	
b	`N/A`	
o	`operationId`	
i	`Service.requestDeposit`	
α	`Service.requestDeposit#1`	
σ	`IYGTManager.requestDeposit`	
c	`IAccountant.transferToExchange r`	
ϕ	`IYGTExchanger.requestDeposit`	
r	`Service.revertDeposit`	

Low-level spec

N	Human description	Formal description
		$\forall\, y \in IYGTManager,\ a, m \in \aleph,\ \varsigma \in Service,\ b \in boolean,\ s_1, s_2 \in address$
		$\$ = 'USDT',\ \$.balanceOf(s_1) = \$.balanceOf(s_2),$
		$\$.allowance(s_1, a) = \$.allowance(s_2, a)$

		`ε = {`
		`  o : N = ς.requestDeposit#1(e)`
		`  IYGTManager(ς.managers[y]).requestDeposit(o, msg.sender, a, m)`
		`  v : N = ς.requestDeposit#2(o, y, a)`
		`}`

		`κ = {`
		`  ε`
		`  IYGTExchanger(y).requestDeposit(o, msg.sender, v, m)`
		`}`

TII.1	Any user can initiate placing a deposit	$\frac{\{msg.sender=s_1\}\varsigma.requestDeposit(y,a,m)\{S=b\}}{\{msg.sender=s_2\}\varsigma.requestDeposit(y,a,m)\{S=b\}}$
TII.2	Placing a deposit is successful when and only when "accept", "store" and "collect" is successful	$\frac{\{\ \}\varepsilon(y,a,m)\{S=b\}}{\{\ \}\varsigma(y,a,m)\{S=b\}}$
TII.3	Operation is nothing beforehand	
TIO.1	The result is either 'send' or 'revert'	$\frac{\{\ \}\varsigma(y,a,m)\{S=true\}}{\varsigma(y,a,m)\Leftrightarrow\kappa(y,a,m)}$
TIO.2	If the result is "send", then : ● fee is collected ● operation is stored ● operation is sent to the bridge	
TIO.3	If the result is "reverted", then	

	the result is reverted	

Scenario TRC20 Deposit Accept

Mapping

Original element	Mapped element	Comments
ss	`N/A`	
s	`msg.sender`	
y	`exchanger`	
a	`value`	
o	`msg`	
o_1	$\texttt{msg}_1$	
A	`Service.requestDeposit#1`	

Low-level spec

N	Human description	Formal description
		$\forall y \in IYGTExchanger,\ \varsigma \in Service$

TAI.1	Scenario can be called by TRC20 Deposit	
TAI.2	Rate is successful if and only if: YTG is known	$\frac{\varsigma.managers[y] \mathrel{!=} null}{\{\ \}\varsigma.requestDeposit\#1(y)\{S=true\}}$
TAO.1	The resulting operation is:	
	created	
	assigned by unique ID	$\frac{\{\ \}\varsigma.requestDeposit\#1(msg,y),\varsigma.requestDeposit\#1(msg_1,y)\left\{S=true\wedge S_1=true\right\}}{\varsigma.requestDeposit\#1(msg,y)=\varsigma.requestDeposit\#1(msg_1,y)\Leftrightarrow msg\equiv msg_1}$
	assigned by amount	Moved to TOO.2a
	assigned by YTG address	Moved to TOO.2a
	assigned by initiator	Moved to TOO.2a
TAO.2	Upon success the upper scenario moves to accepted state	
TAO.3	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Deposit Store

Mapping

Original element	Mapped element	Comments

ss	`YTGManager.service`	
y	`YTGManager`	
a	`value`	
o	`(operationId, minReturnValue)` ⇒ `OperationInfo`	
O	`YTGManager.requestDeposit`	
l.ma	`YTGManager.minDepositValue`	

Low-level spec

N	Human description	Formal description
		$\forall y \in IYGTManager,\ a, m, o \in \aleph,\ u \in address$
TOI.1	Only TRC20 Deposit Initiate can call this scenario	$\{msg.sender \neq y.service\}y.requestDeposit(o, u, a, m)\{S = false\}$
TOI.2	If deposit is too low the action fails	$\{a < y.minDepositValue\}y.requestDeposit(o, u, a, m)\{S = false\}$
TOI.3	Otherwise, the action is successful	$\{msg.sender = u.service \land a \geq y \land minDepositValue \leq y.deposits[o].status = None\}$ $y.requestDeposit(o, u, a, m)\{S = true\}$
TOI.4	Operation is not in operations	$\{y.deposits[o].status \neq None\}y.requestDeposit(o, u, a, m)\{S = false\}$

TOI.5	The scenario is not in stored state before start	
TOO.1	The scenario is in stored state after finish	Merged with TOO.2a
TOO.2	Operation is stored in operations	$\frac{\{\ \}y.requestDeposit(o,u,a,m)\{S=true\}}{\{\ \}y.requestDeposit(o,u,a,m)\{y.deposits[o]=OperationInfo(u,a,m,Pending)\}}$
TOO.3	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Deposit Collect

Mapping

Original element	Mapped element	Comments
ss	`Accountant.service`	
o	`(operationId, user, ygtExchanger, value)`	
c	`Service.requestDeposit#2` $\lor$ `IYGTExchanger.getToken` $\lor$ `IAccountant.transferForExchange`	
l.f	`fee[exchanger].deposit`	
o.a	`return value`	

o.f	`lockedFeesInfo[operationId]`	

Low-level spec

N	Human description	Formal description
		$\forall a \in Accountant,\ o, v \in \aleph, e \in IYGTExchanger,\ u \in address, t \in YGTToken, \$ = 'USDT'$
		$\varsigma \in Service,\ f = \frac{v \cdot a.fee[e].deposit}{FEEFACTOR}$
TCI.1	Only TRC20 Deposit Initiate can call this scenario	$\{msg.sender \neq a.service\}a.transferToExchanger(o, t, u, e, v)\{S = false\}$
TCI.2	In case of low balance action fails	$\{\$.balanceOf(msg.sender) < v\}\varsigma.requestDeposit\#2(o, e, v)\{S = false\}$
TCI.3	In case of low allowance action fails	$\{\$.allowance(msg.sender, a) < v\}\varsigma.requestDeposit\#2(o, e, v)\{S = false\}$
TCI.4	Otherwise, the action is successful	$\{\$.balanceOf(msg.sender) \geq v \land \$.allowance(msg.sender, a) \geq v \land$ $\sigma = e.service \land v \geq e.minDepositValue\}$ $\varsigma.requestDeposit\#2(o, e, v)\{S = true\}$
TCI.5	The scenario is not in collected state before start	
TCI.6	Fee is zero before start	
TCO.1	The scenario is in collected state after finish	

TCO.2	Fee calculated properly	$\frac{\{\ \}\varsigma.requestDeposit\#2(o,e,v)\{S=true\}}{\{\ \}\varsigma.requestDeposit\#2(o,e,v)\{a.lockedFees[e.token]{+}{=}f\}}$
TCO.2a	Operation is fully initialized[5]	$\frac{\{\ \}\varsigma.requestDeposit\#2(o,e,v)\{S=true\}}{\{\ \}r=\varsigma.requestDeposit\#2(o,e,v)\{lockedFeesInfo[o]=LockedFeesInfo(e.token,f,v)\wedge r=v-f\}}$
TCO.3	Sender's balance is decreased by amount	$\frac{\{\ \}\varsigma.requestDeposit\#2(o,e,v)\{S=true\}}{\{\ \}\varsigma.requestDeposit\#2(o,e,v)\{\$..balanceOf(msg.sender){-}{=}v\}}$
TCO.4	Sender's alowance is decreased by amount	$\frac{\{\ \}\varsigma.requestDeposit\#2(o,e,v)\{S=true\}}{\{\ \}\varsigma.requestDeposit\#2(o,e,v)\{\$..allowance(msg.sender,a){-}{=}v\}}$
TCO.5	Exchanger's balance is increased by deposit but fee	$\frac{\{\ \}\varsigma.requestDeposit\#2(o,e,v)\{S=true\}}{\{\ \}\varsigma.requestDeposit\#2(o,e,v)\{\$..balanceOf(e){+}{=}v-f\}}$
TCO.6	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Deposit Send

Mapping

Original element	Mapped element	Comments
ss	`YGTSwftExchanger.service`	
o	`(operationId, user, value, minReturnValue)` `(value, minReturnValue)`	

[5] As mentioned in the corresponding sections, some items from the high-level specification moved here

c	`(YGTSwftExchanger.token, "USDC", YGTSwftExchanger.lendingManager, value, minReturnValue)`	
r	`YGTSwftExchanger.requestDeposit`	
l.b	`YGTSwftExchanger._swapBridge`	

Low-level spec

N	Human description	Formal description
		$\forall e \in IYGTExchanger, o, v, mv \in \aleph,\ u \in address, \$_t = 'USDT', \ \$_e = 'USDC'$
TSI.1	Only 'TRC20 Deposit Initiate' can request the sending	$\{e.service \neq msg.sender\} e.requestDeposit(o, u, v, mv)\{S = false\}$
TSI.2	In this case sending request is successful	$\left\{e.service = msg.sender \wedge v \geq mv \wedge \$_t.balanceOf(e) \geq v\right\}$ $e.requestDeposit(o, u, v, mv)\{S = true\}$
TSI.3	Operation is not registered as being handled before requesting the send	
TSI.4	Only the bridge can initiate the sending continuation	Swft is simpler than Meson, so some requirements are not actual
TSI.5	In this case sending	Swft is simpler than Meson, so some requirements are not actual

	continuation is successful	
TSI.6	Operation is marked as collected beforehand	
TSI.7	In case of negative result the effect of sending continuation is the same as for timeout	Swft is simpler than Meson, so some requirements are not actual
TSI.8	Only the info bridge can call timeout	Swft is simpler than Meson, so some requirements are not actual
TSI.9	In this case the call is successful	Swft is simpler than Meson, so some requirements are not actual
TSI.10	Operation is registered as being handled before sending continuation	Swft is simpler than Meson, so some requirements are not actual
TSI.11	Operation is registered as being handled before timeout	Swft is simpler than Meson, so some requirements are not actual
TSI.12	Operation's creation data is not created before sending continuation	Swft is simpler than Meson, so some requirements are not actual
TSO.1	Assets are sent to the bridge	$\frac{e.requestDeposit(o,u,v,mv)\{S=true\}}{\left\{\forall x\in\aleph: x=\$_t.balanceOf(e)\right\}e.requestDeposit(o,u,v,mv)\left\{\$_t.balanceOf(e)=x-v\right\}}$ $\frac{e.requestDeposit(o,u,v,mv)\{S=true\}}{\left\{\forall x\in\aleph: x=\$_t.balanceOf(e.bridgeVault)\right\}e.requestDeposit(o,u,v,mv)\left\{\$_t.balanceOf(e.bridgeVault)=x+v\right\}}$
TSO.2	Nothing else is changed	Specified directly at Stage 7

TSO.3	Operation is registered as being handled after sending request	Swft is simpler than Meson, so some requirements are not actual
TSO.4	Nothing else is changed	Swft is simpler than Meson, so some requirements are not actual
TSO.5	In case of timeout (or negative sending continuation): ● Operation is unregistered from handling list ● Operation is marked as rejected	Swft is simpler than Meson, so some requirements are not actual
TSO.6	Nothing else is changed	Swft is simpler than Meson, so some requirements are not actual

Scenario TRC20 Deposit Result

Mapping

Original element	**Mapped element**	**Comments**
s	`msg.sender`	
o	`(operationId,value)`	
h	`YGTSwftExchanger.confirmDeposit`	
l.i	`UGTSwftExchanger.owner`	

Low-level spec

N	Human description	Formal description
		$\forall e \in IYGTExchanger, o, t \in \aleph,\ s = e.service$

		`c(o,t) = {`
		`  managers[e].confirmDeposit(o,t)`
		`  s.accountant.confirmDeposit(o)`
		`}`

THI.1	Only info bridge can initiate this scenario	$\{e.owner \neq msg.sender\}e.confirmDeposit(o,t)\{S = false\}$
		$s.confirmDeposit(o,t) = c(o,t)$
THI.2	Initial status states for Nothing	▨
THO.1	In case of positive success the positive branch of subscenarios is applied	$\frac{\{\ \ \}e.confirmDeposit(o,t)\{S=true\}}{e.confirmDeposit(o,t)=s.confirmDeposit(o,t)}$
THO.2	In case of negative success the negative branch of subscenarios is applied	▨
THM.1	The operation becomes pre-informed after 'pre-inform'	▨

THM.2	The operation becomes informed after 'inform'	▨
THM.3	The operation becomes either minted or rejected after 'mint'	▨
THM.4	The operation becomes accepted after 'mint'	▨
THM.5	The operation becomes completed after 'accept'	▨
THM.6	The operation becomes pre-reverted after 'reject'	▨
THM.7	The operation becomes reverted after 'pre-revert'	▨

Scenario TRC20 Deposit Pre-Inform

This scenario is fully covered by TRC20 Deposit Result.

Scenario TRC20 Deposit Inform

This scenario is fully covered by TRC20 Deposit Result.

Scenario TRC20 Deposit Mint

Mapping

Original element	Mapped element	Comments
ss	`YGTManager.service`	
o	`(operationId, shares) YGTManager.deposits[operationId]`	

Low-level spec

N	Human description	Formal description
		$\forall e \in YGTSwftExchanger, o, t \in \aleph,\ b \in boolean$
		$s = e.service,\ m = s.managers[e],\ O = m.deposits[o]$
TMI.1	The scenario is successful if and only if: • It initiated by 'TRC20DepositResult' • YTG is known • Operation has been registered • Operation amount is positive • Required shares number is positive	$\{b = (msg.sender = m.service \wedge\ m \neq \varnothing \wedge\ O \neq \varnothing \wedge\ O.amount > 0) \wedge t > 0 \wedge o.status = P\}$ $m.confirmDeposit(o, t)$ $\{S = b\}$
TMI.2	If the scenario is successful, it goes to the 'mint' branch	

TMI.3	If the scenario is successful and negative, it goes to the 'revert' branch	
TMI.4	Operation is in 'Informed' stage beforehand	Moved to TMI.1
TMO.1	If the scenario is successful and positive: ● YTG balance of the initiator is increased accordingly: ● The status is changed to 'Minted' ● The operation is removed from the list	$\frac{\{\ \}m.confirmDeposit(o,t)\{S=true\}}{\{\forall \delta \in \aleph : \delta = e.token.balanceOf(O.user)\}m.confirmDeposit(o,t)\{e.token.balanceOf(O.user) = \delta + t \wedge O.status = CONFIRMED\}}$
TMO.2	Nothing else is changed	Specified directly at Stage 7
TMO.3	If the scenario is successful and negative: ● The status is changed to 'Rejected' ● The operation is removed from the list	Not actual for the current version. Reverting is identified and called externally
TMO.4	Nothing else is changed	Not actual for the current version. Reverting is identified and called externally

Scenario TRC20 Deposit Accept

This scenario is fully covered by TRC20 Deposit Result

Scenario TRC20 Deposit Complete

Mapping

Original element	Mapped element	Comments
ss	`msg.sender`	
o	`operationId` `lockedFeesInfo[operationId]`	
c	`Accountant.confirmDeposit`	

Low-level spec

N	Human description	Formal description
		$\forall a \in Accountant, o \in \aleph, b \in boolean, \chi = a.lockedFees[a.lockedFeesInfo[o].token]$
TZI.1	The action is successful if and only if: ● action is initiated by 'TRC20 Deposit Result' ● Operation is registered as being handled	$\{b = (a.service = msg.sender \land a.lockedFeesInfo[o].token \neq \emptyset)\}$ $a.confirmDeposit(o)$ $\{S = b\}$
TZI.2	The status of the operation is	▨

	'Accepted' beforehand	
TZO.1	The status of the operation is 'Completed' afterwards	
TZO.2	Operation is not registered as being handled afterwards	$\frac{\{\ \}a.confirmDeposit(o)\{S=true\}}{\{\ \}a.confirmDeposit(o)\{a.lockedFeesInfo[o]=\emptyset\}}$
TZO.3	Balance of the Accountant is decreased by the fee	$\frac{\{\ \}a.confirmDeposit(o)\{S=true\}}{\{\forall f{:}f=\chi\}a.confirmDeposit(o)\{\chi=f-a.lockedFeesInfo[o].fees\}}$
TZO.4	Balance of the Storage is increased by the fee	
TZO.5	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Deposit Reject

This scenario is fully abandoned, as reverting is called off-chain

Scenario TRC20 Deposit Revert

Mapping

Original element	**Mapped element**	**Comments**
ss	`msg.sender`	
o	`operationId`	

R	`YGTSwftExchanger.revertDeposit`	

Low-level spec

N	Human description	Formal description
		$\forall e \in YGTSwftExchanger, o, v \in \aleph,\ b \in boolean,\ \chi = a.lockedFees[a.lockedFeesInfo[o].token]$
		$s = e.service,\ m = s.managers[e],\ a\ =\ s.accountant, b \in boolean.\ \$ = 'USDT'$
		$u = m.deposits[o[.addr$
TYI.1	The action is successful if and only if the initiator is 'TRC20 Deposit Result'	$\{b = (msg.sender = e.owner \wedge e.token.balanceOf(e) \geq v \wedge m \neq \varnothing \wedge m.deposits[o] \neq \varnothing \wedge$ $m.deposits[o].amount > 0 \wedge m.deposits[o].amount \geq v \wedge$ $a.lockedFeesInfo[o[\neq \varnothing \wedge m.deposits[o].status = PENDING)\}$ $e.revertDeposit(o, v)$ $\{S = b\}$
TYI.2	The operation is in 'Pre-Reverted' stage beforehand	Moved to TYI.1
TYO.1	In case of success the action the initiator gets the full money back	$\frac{\{\ \ \}e.revertDeposit(o,v)\{S=true\}}{\{\forall \delta \in \aleph : \delta = \$.balanceOf(u)\}e.revertDeposit(o,v)\{\$.balanceOf(u) = \delta + v\}}$
TYO.2	Fee is returned from the accountant	$\frac{\{\ \ \}e.revertDeposit(o,v)\{S=true\}}{\{\forall \delta \in \aleph : \delta = \chi\}e.revertDeposit(o,v)\{\chi = \delta - a.lockedFeesInfo[o].fees\}}$
TYO.3	The main amount is returned from the exchanger	$\frac{\{\ \ \}e.revertDeposit(o,v)\{S=true\}}{\{\forall \delta \in \aleph : \delta = \$.balanceOf(e)\}e.revertDeposit(o,v)\{\$.balanceOf(e) = \delta - v + a.lockedFeesInfo[o].fees\}}$

TYO.4	The operation is moved to the reverted stage	$\frac{\{\ \}e.revertDeposit(o,v)\{S=true\}}{\{\ \}e.revertDeposit(o,v)\{m.deposits[o].status=REVERTED\}}$
TYO.5	The operation is removed from the operation list	$\frac{\{\ \}e.revertDeposit(o,v)\{S=true\}}{\{\ \}e.revertDeposit(o,v)\{m.deposits[o]=\varnothing\}}$
TYO.6	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Withdrawal Initiate

Mapping

Original element	Mapped element	Comments
s	`msg.sender`	
y	`Service.exchanger`	

o	`(value,minReturnValue)`
	`(operationId, value, minReturnValue)`	

i	`Service.requestWithdrawal`	

Low-level spec

N	Human description	Formal description
		$\forall s_1, s_2 \in A,\ e \in IYGTExchanger,\ v, mv \in \aleph,\ \varsigma \in Service, b \in boolean$

		$e.token.balanceOf(s_1) = e.token.balanceOf(s_2)$

		`r(e,v,mv)= {`
		`  o = ς.requestWithdrawl#1(e)`
		`  s = IYGTManager(ς.managers[e].requestWithdrawal(o,msg.sender,v,mv))`
		`  ς.requestWithdrawl#2(o,e,s,mv)`
		`}`

WII.1	Anybody can initiate the scenario	$\frac{\{msg.sender=s_1\}ς.requestWithdrawl(e,v,mv)\{S=b\}}{\{msg.sender=s_2\}ς.requestWithdrawl(e,v,mv)\{S=b\}}$
WII.2	Operation is not created beforehand	▨
WIM.1	Operation is created and accepted after 'accept'	▨
WIM.2	Operation is burnt after 'burn'	▨
WIM.3	Operation is either sent or rejected after 'request'	▨
WIM.4	Operation is reverted after 'revert'	▨
WIO.1	If the scenario is successful and the operation ended up with the 'sent' status, it means it's a composition of 'accept', 'burn' and 'request'	$\frac{\{\ \}ς.requestWithdrawl(e,v,mv)\{S=true\}}{ς.requestWithdrawl(e,v,mv)=r(e,v,mv)}$

Scenario TRC20 Withdrawal Accept

Mapping

Original element	Mapped element	Comments
o	`(exchanger,value,minValue)`	
a	`value`	
α	`Service.requestWithdrawal#1`	

Low-level spec

N	Human description	Formal description
		$\forall\, e \in IYGTExchanger,\ v, mv \in \aleph,\ \varsigma \in Service, b \in boolean$
WAI.1	The scenario can be initiated by 'TRC20WithdrawalInitiate' scenario only	▨
WAI.2	The token must be registered beforehand	$\{\varsigma.managers[e] = \emptyset\}\varsigma.requestWithdrawal\#1(e)\{S = false\}$
WAI.3	Operation is not created beforehand	▨
WAI.4	Otherwise, the operation is successful	$\{\varsigma.managers[e] \neq \emptyset\}\varsigma.requestWithdrawal\#1(e)\{S = true\}$

WAO.1	In case of success: ● Operation is created ● Amount and token are assigned ● Operation status is changed to 'accepted'	
WAO.2	Operation is unique	$\frac{\{\ \}\varsigma.requestWithdrawal\#1(msg,e),\varsigma.requestWithdrawal\#1(msg_1,e)\left\{S=true\wedge S_1=true\right\}}{\varsigma.requestWithdrawal\#1(msg,e)=\varsigma.requestWithdrawal\#1(msg_1,e)\Leftrightarrow msg\equiv msg_1}$
WAO.3	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Withdrawal Burn

Mapping

Original element	**Mapped element**	**Comments**
ss	`YGTManager.service`	
o	`(operationId, value, minValue) withdraw`	
b	`YGMManager.requestWithdrawal`	
o.a	`v`	
l.ma	`YGTManager.minWithdrawValue`	

Low-level spec

N	Human description	Formal description
		$\forall\, m \in YGTManager,\ o, v, mv \in \aleph,\ \varsigma \in Service, u \in address$
		$\sigma = min\left(\frac{v \cdot m.token.oracle.shares}{m.token.oracle.pool}, m.token.sharesOf(u)\right)$
		$\frac{v \cdot m.token.oracle.shares}{m.token.oracle.pool} \geq m.token.sharesOf(u) \Rightarrow \omega = \frac{m.token.sharesOf(u) \cdot m.token.oracle.pool}{m.token.oracle.shares}$
		$\frac{v \cdot m.token.oracle.shares}{m.token.oracle.pool} < m.token.sharesOf(u) \Rightarrow \omega = v$
WBI.1	The scenario must be initiated by the 'TRC20 Withdrawal Initiate' scenario	$\{m.service \neq msg.sender\}m.requestWithdrawal(o, u, v, mv)\{S = false\}$
WBI.2	The scenario fails if the amount is less than minimally allowed	$\{\omega < m.minWithdrawValue\}m.requestWithdrawal(o, u, v, mv)\{S = false\}$
WBI.3	The scenario fails if the user does not have any YTG tokens	$\{m.token.sharesOf(u) = 0\}m.requestWithdrawal(o, u, v, mv)\{S = false\}$
WBI.4	In this case the scenario is successful	$\{m.service = msg.sender \wedge \omega \geq m.minWithdrawValue \wedge m.token.sharesOf(u) > 0$ $m.withdrawals[o] \neq \varnothing \wedge u > 0\}$ $m.requestWithdrawal(o, u, v, mv)\{S = true\}$
WBI.5	The operation is in 'accepted' state beforehand	

WBO.1	The operation is in 'burnt' state afterwards	
WBO.2	The balance of the sender is decreased accordingly	$\frac{\{\ \}m.requestWithdrawal(o,u,v,mv)\{S=true\}}{\{\forall x\in\aleph:x=m.token.sharesOf(u)\}m.requestWithdrawal(o,u,v,mv)\{m.token.sharesOf(u)=x-\sigma\}}$
WBO.3	The balance of the manager is increased accordingly	Deprecated
WBO.4a	The shares amount of the operation is assigned	$\frac{\{\ \}m.requestWithdrawal(o,u,v,mv)\{S=true\}}{\{\ \}r=m.requestWithdrawal(o,u,v,mv)\{r=\sigma\}}$
WBO.4b		$\frac{\{\ \}m.requestWithdrawal(o,u,v,mv)\{S=true\}}{\{\ \}r=m.requestWithdrawal(o,u,v,mv)\{m.withdrawals[o]=OperationInfo(u,\sigma,mv,PENDING)\}}$
WBO.5	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Withdrawal Request

Mapping

Original element	Mapped element	Comments
ss	`YGTManager.service`	
o	`(operationId,shares,minReturnValue)`	
r	`Service.requestWithdrawal#2` `YGTSwftExchanger.requestWithdrawal`	

Low-level spec

N	Human description	Formal description
		$\forall\ e\ \in\ YGTSwftExchanger,\ o, v, mv, \sigma\ \in\ \aleph,\ \varsigma\ \in\ Service,\ s\ \in\ address$
		$\mu\ =\ \frac{v\cdot\varsigma.accountant.fee[e].withdrawal}{FEEFACTOR-\varsigma.accountant.fee[e].withdrawal}$
WRI.1	Request can be initiated by the 'TRC20 Withdrawal Initiate' scenario only	$\{e.service\ \neq\ msg.sender\}e.requestWithdrawl(o, s, \sigma, \mu)\{S\ =\ false\}$
WRI.2	If so, the request is successful	$\{e.service\ =\ msg.sender\}e.requestWithdrawl(o, s, \sigma, \mu)\{S\ =\ true\}$
WRI.3	The operation is in 'burnt' state before any action	
WRI.4	The operation is not in the info bridge list before the request	
WRI.5	Response can be initiated by the info bridge only	Swft is simpler than Meson, so some requirements are not actual
WRI.6	If so, the response is successful	Swft is simpler than Meson, so some requirements are not actual
WRI.7	The operation is in the info bridge before the response	Swft is simpler than Meson, so some requirements are not actual
WRO.1	As a result of the ‘request’ operation is registered by the info bridge	

WRO.2	Nothing else is changed	Specified directly at Stage 7
WRO.3	In case of the positive result, the operation status is changed to 'sent'	
WRO.4	Nothing else is changed	
WRO.5	In case of the negative result, the operation status is changed to 'rejected'	Swft is simpler than Meson, so some requirements are not actual
WRO.6	In case of the negative result, the operation is unregistered from the info bridge	Swft is simpler than Meson, so some requirements are not actual
WRO.7	Nothing else is changed	Swft is simpler than Meson, so some requirements are not actual

Scenario TRC20 Withdrawal Result

Mapping

Original element	Mapped element	Comments
s	`msg.sender`	
o	`(operationId, value)`	
r	`YGTSwftExchanger.requestWithdrawal` `Service.requestWithdrawal`	

Low-level spec

N	Human description	Formal description
		$\forall\ e \in YGTSwftExchanger,\ o, v, \in\ \aleph,\ \varsigma \in Service, \varsigma = e.service$

		`ec(o,v) = {`
		`  e.confirmWithdrawal#1(v)`
		`  e.service.confirmWithdrawal(o,v)`
		`}`

		`sc(e,o,v) = {`
		`  u = ς.managers[msg.sender]).confirmWithdrawal(o,v)`
		`  ς.confirmWithdrawal#1(u,v)`
		`}`

ERI.1	The scenario can be initiated by the info bridge only	$\{e.owner \neq msg.sender\}e.confirmWithdrawal(o, v)\{S = false\}$
ERI.1a		$\{\varsigma.managers[msg.sender]\ = \varnothing\}\varsigma.confirmWithdrawal(o, v)\{S = false\}$
ERI.2	If so, the scenario is successful	$\{e.owner = msg.sender \wedge \varsigma.managers[msg.sender].deposits[o].status = Pending \wedge \varsigma.accountant \neq 0 \wedge v > 0 \wedge \$.b(msg.sender) \geq v \wedge \$.a(msg.sender, \varsigma.accountant) \geq v\}$ $e.confirmWithdrawal(o, v)\{S = true\}$
ERI.2a		$\{\varsigma.managers[msg.sender] \neq \varnothing\ \wedge \varsigma.managers[msg.sender].deposits[o].status = Pending \wedge \varsigma.accountant \neq 0 \wedge v > 0 \wedge \$.b(msg.sender) \geq v \wedge \$.a(msg.sender, \varsigma.accountant) \geq v\}$ $\varsigma.confirmWithdrawal(o, v)\{S = true\}$
ERI.3	Operation status is 'nothing' beforehand	

ERM.1	Operation status is 'received' after Receive	
ERM.2	Operation status is 'informed' after Inform	
ERM.3	Operation status is either 'accepted' or 'rejected' after Manage	
ERM.4	Operation status is 'assigned' after Assign	
ERM.5	Operation status is 'completed' after Complete	
ERM.6	Operation status is 'reverted' after Revert	
ERO.1	In case of a positive result, the final result is a composition of the corresponding functions	$\frac{\{\ \}e.confirmWithdrawal(o,v)\{S=true\}}{e.confirmWithdrawal(o,v)=ec(o,v)}$
ERO.1a		$\frac{\{\ \}\varsigma.confirmWithdrawal(o,v)\{S=true\}}{\varsigma.confirmWithdrawal(o,v)=sc(o,v)}$

Scenario TRC20 Withdrawal Receive

Mapping

Original element	Mapped element	Comments

o	`(operationId, value)`	
ϱ	`YGTSwftExchanger.confirmWithdrawal#2`	
I.e	`YGTSwftExchanger`	
I.a	`YGTSwftExchanger.service.accountant`	

Low-level spec

N	Human description	Formal description
		$\forall\ e \in YGTSwftExchanger,\ o, v \in \aleph, \$ = 'USDT'$
WVI.1	The scenario is successful if the initiator is 'TRC20 Withdrawal Result' only	
WVI.2	If so, the scenario is successful	$\{e.service.accountant > 0 \land e.owner = msg.sender\}e.confirmWithdrawal\#1(o, v)\{S = true\}$
WVI.3	The operation is in 'Nothing' status beforehand	
WVO.1	Operation status is turned to 'Received'	
WVO.2	Allowance from Exchanger to Accountant is set to the	$\dfrac{\{\ \}e.confirmWithdrawal\#1(o,v)\{S=true\}}{\{\ \}e.confirmWithdrawal\#1(o,v)\{\$.allowanceOf(e,e.service.accountant)=v\}}$

	received value	
WVO.3	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Withdrawal Inform

This scenario is fully covered by TRC20 Withdrawal Result

Scenario TRC20 Withdrawal Manage

Mapping

Original element	Mapped element	Comments
ss	`YGTManager.service`	
o	`(operationId,value)` `YGTManager.withdrawals`	
m	`YGTManager.confirmWithdrawal`	

Low-level spec

N	Human description	Formal description
		$\forall\, m \in YGTManager,\ o, v \in \aleph$
WMI.1	The scenario can be initiated by the 'TRC20 Withdrawal	$\{m.service \neq msg.sender\}m.confirmWithdrawal(o, v)\{S = false\}$

	Result' only	
WMI.2	The operation must be in the list of operations	$\{m.withdrawals[o] = \emptyset\}m.confirmWithdrawal(o, v)\{S = false\}$
WMI.3	If the returned amount is zero, the operation fails	$\{v = 0\}m.confirmWithdrawal(o, v)\{S = false\}$
WMI.4	The operation is in 'Informed' state beforehand	$\{m.withdrawals[o].status \neq PENDING\}m.confirmWithdrawal(o, v)\{S = false\}$
WMI.5	If so, the scenario is successful	$\{m.service = msg.sender \wedge m.withdrawals[o] \neq \emptyset \wedge v > 0 \wedge m.withdrawals[o].status = P\}$ $m.confirmWithdrawal(o, v)\{S = true\}$
WMO.1	If the scenario is positive, the operation is turned into 'Accepted' state	$\frac{\{\ \}m.confirmWithdrawal(o,v)\{S=true\}}{\{\ \}m.confirmWithdrawal(o,v)\{m.withdrawals[o].status=CONFIRMED\}}$
WMO.2	The locked amount is no longer belongs to the Manager	Deprecated
WMO.3	The exchanged tokens are burnt	Deprecated
WMO.4	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Withdrawal Assign

This scenario is fully covered by TRC20 Withdrawal Initiate

Scenario TRC20 Withdrawal Complete

Mapping

Original element	Mapped element	Comments
ss	`Accountant.service`	

o	`(u,value)`
	`(token,ygtExchanger,user,value)`	

c	`Service.confirmWithdrawal#2` `Accountant.withdrawFromExchanger`	

Low-level spec

N	Human description	Formal description
		$\forall\, \varsigma \in Service,\ a \in Accountant,\ o, v \in \aleph, e \in IYGTExchanger,\ u \in address,\ t \in YGTToken$
		$b \in boolean,\ \$ = 'USDT'$
		$f = \frac{v \cdot \varsigma.accountant.fee[msg.sender].withdrawal}{FEEFACTOR}$
WLI.1	The action is successful when and only when it's initiated by 'TRC20WithdrawalResult scenario'	$\{b = (a.service = msg.sender)\}a.withdrawFromExchanger(t, e, u, v)\{S = b\}$

WLI.2	Operation is 'assigned' beforehand	
	If the action is successful:	
WLO.1	Operation becomes 'completed'	
WLO.2	Operation is removed from the list	
WLO.3	USDC balance of the exchanger is decreased by the amount	$\frac{\{\ \}\varsigma.confirmWithdrawal\#1(u,v)\{S=true\}}{\{\forall x\in\aleph:x=\$..balanceOf(msg.sender)\}\varsigma.confirmWithdrawal\#1(u,v)\{\$..balanceOf(msg.sender)=x-v\}}$
WLO.4	USDC balance of the accountant is increased by the fee	$\frac{\{\ \}\varsigma.confirmWithdrawal\#1(u,v)\{S=true\}}{\{\forall x\in\aleph:x=\$..balanceOf(\varsigma.accountant)\}\varsigma.confirmWithdrawal\#1(u,v)\{\$..balanceOf(\varsigma.accountant)=x+f\}}$
WLO.5	USDC balance of the initiator is increased by the amount subtracted by fee	$\frac{\{\ \}\varsigma.confirmWithdrawal\#1(u,v)\{S=true\}}{\{\forall x\in\aleph:x=\$..balanceOf(u)\}\varsigma.confirmWithdrawal\#1(u,v)\{\$..balanceOf(u)=x+v-f\}}$
WLO.6	Nothing else is changed	Specified directly at Stage 7

Scenario TRC20 Withdrawal Revert

Mapping

Original element	Mapped element	Comments

s	`msg.sender`	
o	`operationId` `withdrawals[operationId]`	
r	`YGTSwftExchanger.revertWithdrawal` `Service.revertWithdrawal` `YGTManager.revertWithdrawal`	

Low-level spec

N	Human description	Formal description
		$\forall\, \varsigma \in Service,\ m \in YGTManager,\ o, v, \sigma \in \aleph, e \in YGTSwftExchanger,\ u \in address$
		$t \in YGTToken, \$ = 'USDT'$
		$\varsigma = e.service,\ m = \varsigma.managers[e]$
TEI.1	The scenario can be initiated by InfoBridge only	$\{e.owner \neq msg.sender\} e.revertWithdrawal(o, \sigma)\{S = false\}$
TEI.1a		$\{\varsigma.managers[msg.sender] = \varnothing\} \varsigma.revertWithdrawal(o, \sigma)\{S = false\}$
TEI.1b		$\{m.service \neq msg.sender\} m.revertWithdrawal(o, \sigma)\{S = false\}$
TEI.2	If so, the scenario is successful	$\{m.service = msg.sender \wedge m.withdrawals[o] \neq \varnothing \wedge m.withdrawals[o].status = PENDING\}$ $m.revertWithdrawal(o, \sigma)\{S = true\}$
TEI,3	The operation is in 'rejected'	$\{m.withdrawals[o].status \neq PENDING\} m.revertWithdrawal(o, \sigma)\{S = false\}$

	state beforehand	
TEI.4	Operation is enlisted beforehand	$\{m.withdrawals[o] = \emptyset\} m.revertWithdrawal(o, \sigma) \{S = false\}$
TEO.0		$\frac{\{\ \} e.revertWithdrawal(o,\sigma) \{S=true\}}{e.revertWithdrawal(o,\sigma) \equiv \varsigma.revertWithdrawal(o,\sigma)}$
TEO.0a		$\frac{\{\ \} e.revertWithdrawal(o,\sigma) \{S=true\}}{e.revertWithdrawal(o,\sigma) \equiv m.revertWithdrawal(o,\sigma)}$
TEO.1	The operation is in 'reverted' state afterwards	$\frac{\{\ \} e.revertWithdrawal(o,\sigma) \{S=true\}}{\{\ \} e.revertWithdrawal(o,\sigma) \{m.withdrawals[o].status=REVERTED\}}$
TEO.2	The operation is delisted afterwards	
TEO.3	The Manager token balance is reverted to the original state	Deprecated
		$u = m.withdrawals[o].addr$
TEO.4	The user token balance is reverted to the original state	$\frac{\{\ \} e.revertWithdrawal(o,\sigma) \{S=true\}}{\{\forall x \in \aleph : x = e.token.sharesOf(u)\} e.revertWithdrawal(o,\sigma) \{e.tokens.sharesOf(u) = x + \sigma\}}$
TEO.5	Nothing else is changed	Specified directly at Stage 7

Verification

The verification code is placed at https://github.com/Pruvendo/molecula. 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
- Conversion to Eval/Execs
- Joining the formal verification with the implementation(Eval/Execs)
- Interpreting the formal verification outcome

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

Severity description

All the issues and notes have an assigned severity, identified by their background colors. The following severities and colors are in place:

- Critical - possibility of theft of assets, freezing of assets, ruining the data structure, disclosure of confidential information, etc. Examples are:
    - Ability for unauthorized access
    - Acceptance of invalid requests from users
- Major - possibility for incorrect system behavior preventing the user from performing their goals, but without losing their assets. Examples are:
    - Refusal to accept correct user's requests
    - Periodical refusal to accept correct user's requests
    - Major performance issues

    - Major gas overconsumption
- Minor - any minor issues, such as:
    - Possibility of exotic errors from the system owner
    - Incorrect diagnostics for user's or owner's wrong behavior
    - Minor performance issues
    - Minor gas overconsumption
- Style - all the coding style issues, as described in the corresponding section

Smart contracts (formal verification)

Location	Issue	Status
Commit b607359b519a545ca02be1342cb7abd929860147, on 03/22/24		
Accountant/*setFee*	*_FEE_FACTOR* must be more than *depositFee* and *withdrawalFee* to avoid division by zero	FIXED
Throughout the project	While usage of `revert` is understandable (as custom errors are used) it's hard to distinguish reverts as requires from reverts as assertions. For example, different prefixes can be used	Waived
Exchangers (fields)	The name `exchangers` looks misleading as de facto it stays for **`IYGTManager`**.	FIXED
Exchangers/ *delExchanger*	Loss of assets can happen if the exchanger(manager) is deleted while still handling some transactions?	Waived (this issue will be handled by off-software actions)
Commit 2e2fb234cd68ebdae1a4777f744bc9d59c0eb11b, made on 04/25/24		
Oracle/ *set* methods	No check if the values are positive (and higher some minimal limit). Division by zero or integer overflow potentially may happen	Waived (checked at the web application level)
YTGManager/ *confirmDeposit*	Pending state is not verified (it's minor as a violation must never happen)	FIXED

YTGManager/ *confirmWithdrawal*	Pending state is not verified (it's minor as a violation must never happen)	FIXED
Oracle	External update of pool and shares looks potentially dangerous	Waived (as some off-chain part is supposed to exist anyway, its proportion is not critical). Anyway, Pruvendo recommends to put maximal possible part of business logic into smart contracts to prevent system malfunction
YTGManager/ *revertWithdrawal*	Pending state is not verified (it's minor as a violation must never happen)	FIXED
YTGManager/ *revertWithdrawal*	It's suggested to introduce something like *transferShares* in *YTGToken* to make the logic more straightforward	Waived

Oracle (classic audit)

- There were three iterative audits against the following commits, their hashes and dates are provided below, the composite set of results is provided below:
  - eae6afa2f8e63dcb974e9b281b5e18e2bcbd54a5, made on 03/25/24
  - 1dc4859e8ecd6ee742752ed97e0beb372bb46d68, made on 04/05/24
  - 2db584304d08289c0f6cfa8db2be408202a6e56c, made on 05/08/24
- The following staff was out of consideration:
  - Deployment scripts
  - Web pages
  - Tests, as well as software intended inclusively for testing
  - Meson staff
  - Manual bridging
- For the rest of the software the principles described in the corresponding section are respected and followed
- The list of notes is provided below (there are missing items because they were originally introduced and removed)

N	File	Function	Priorit y	Description	Decision

	global				
0			Major	Some transactions may fail due to lack of gas. No warnings were seen (suggested: Telegram bot and emergency pool). For instance, “smart” ethers that take gas from the emergency pool in case of failure and send an event to the bot	Waived(planned to be implemented via Discord)
0a			Style	A lot of unused code. It’s recommended to clean it up before the release Example is MemoStorage	New
	common				
	blockchain-utilities				
2a	waitForTronTransaction.ts	waitForTronTransaction	Style	Misspelling in comments (“bellow”)	Accepted, but waived as minor
2a	waitForEthereumTransaction.ts	waitForEthereumTransaction	Style	Misspelling in comments (“bellow”)	Accepted,but waived as minor
	<u>utilities</u>				
3	ArgsToString.ts	argsToString	Minor	As `BigInt` is widely used throughout the project, it’s important to mention that `JSON.stringi`	Fixed

				`fy` does not work for it and needs special handling. It's recommended to consider a special case	
4	`Async.ts`	`makeAsync`	`Style`	For `operation` type `any` is used, while the explicit type is possible	Not used in Molecula
5	`Async.ts`	`makeAsyncRev`	`Style`	The same	Not used in Molecula
6	`CommonStore.ts`	`unload`	`Style`	Two nested `if`'s are in place	Not used in Molecula
7	`Log.ts`	`info`	`Minor`	The method is always enabled	Accepted, but waived as minor
8	`Log.ts`	`warning`	`Minor`	The same	Accepted, but waived as minor
9	`SubscriptionCenter.ts`	`subscribe`	`Minor`	Unlike in other cases, there is no defense against a double subscription	Accepted, but waived as minor
12	`calculateAndLocatePercentage.ts`	`calculateAndLocalePercentage`	`Style`	The constant `BigNumber(0)` is widely used throughout the project. It's advised to explicitly define it as a separate constant	Accepted, but waived as minor
13	`stringifyValueToSign.ts`	`stringifyValueToSign`	`Minor`	As before, it does not work for `BigInt`	Fixed
14	`MemoCollectionLoader.ts`	`subscribe`	`Style`	It's usually considered that any function has to be fit within a screen. It's not a case here	Not used in Molecula

15	MemoDocumentLoader.ts	subscribe	Style	The same as above	Not used in Molecula
	middleware				
	evm-deposit-manager				
16	index.ts	deposit	Style	Too long function	Deprecated
17	module.ts	connect	Style	Too long function	Deprecated
18	module.ts	connect	Style	Decision based on message text looks potentially dangerous	Deprecated
20	module.ts	nativeBalance	Style	Getting a provider looks like an expensive operation (taking into account that even division was optimized above). What about having it as a class field	Deprecated
21	module.ts	depositBalance	Style	May be it's better to handle error code rather than response body	Deprecated
22	module.ts	deposit	Critical	`sendTransaction` does not return the confirmed transaction, only the hash. Even more, it's required about 30 sec to get the transaction flown	Deprecated
23	module.ts	deposit	Major	`No contract error is handled`	Deprecated
	tron-deposit-manager				

25	sendAndAwaitDeposit.ts	sendAndAwaitDeposit	Style	Huge function, suggested to be splitted	Refactored
26	sendAndAwaitWithdrawal.ts	sendAndAwaitWithdrawal	Style	Huge function, suggested to be splitted	Refactored
28	sendAndAwait[DEsotitWithdrawal]*.ts	sendAndAwait[DEsotitWithdrawal]*	Major	If the sendRawTransaction returns false, the failure looks unhandled	New
	backend				
	tron-exchanger-evm-sfwt *(lithium)*				
34	deposit.ts	depositRoute	Minor	no check that value is positive	Fixed
35	deposit.ts	depositRoute	Style	function is extremely long	Waived, as minor
36	checkAuth.ts	checkAuth	Minor	Probably it's better to convert everything to lower (upper) case to avoid bugs with cases in future	Fixed
	packages				
38	LendingMagagerSwft.ts	listenToDeposit	Style	It's better to move the method signature into a separate constant	Waived, as minor
39	LendingMagagerSwft.ts	listenToWithdraw	Style	The same as above	Waived, as minor
40	SwftSwap.ts	listenToSwap	Style	The same as above	Waived, as minor

Pool Manager

The Version 1.1 of the present report is intended to introduce Pool Manager as a new entity.

Business level specification update

Purpose

The following new features were implemented:

- Support for multiple tokens:
  - A whole bunch of ERC-20 and ERC-4626 tokens is supported
  - All the supported tokens must be stable and bound to USD
  - All the deposited tokens must also be stable and bound to USD
- Fee taken in favor of system owner from customer's profit
  - Fee is a fixed percent from customer's profit
  - Fee are taken at withdrawal
  - Fee are not immediately transferred to the owner's wallets, the physical withdrawal can occur any time later

Formulae

- $f = \alpha p$, where *f* - fee assigned for the particular withdrawal, α - fee factor (must be less than 1), *p* - customer's profit
- $F = \sum_{i \in W} f_i - \sum_{j \in R} r_j$, where *F* - total fee available for transferring to owner's wallets, *W* - set of all withdrawals, *R* - set of all transfers to owner's wallets, f_i - fee for *i*th withdrawal, r_j - amount of *j*th transfer. Thus:
  - $\Delta_W F = f$, where $\Delta_W F$ - change of available fee for any withdrawal operation
  - $\Delta_R F = - r$, where $\Delta_R F$ - change of available fee for any transfer operation
- $\alpha = \frac{\beta}{\beta_0}$, where β - fee factor, normalized by β_0, so:
  - $f = \frac{\beta p}{\beta_0}$
- $p = v - v_i$, where *v* - customer's value at withdrawal, v_i - customer's value of the amount to withdraw in case of 'nothing to do' (customer's value in case he preferred not to deposit and buys the same amount of shares as available at withdrawal time)
  - *v* is provided by the Oracle
  - $f = \frac{\beta(v - v_i)}{\beta_0}$
- Shares are the entities (liquid tokens) that are provided to the customers at deposit time and burned at withdrawal time. The price of shares at withdrawal time is calculated by the following formula:

- $\rho = \frac{t}{s}$, where *t* - total amount of USD-stable **deposited**, *s* - total amount of shares available

- So:
  - $v_i = \sigma\rho = \frac{\sigma t}{s}$, where σ - amount of shares received at deposit time
  - $f = \frac{\beta(vs - \sigma t)}{s(\beta_0 - \beta)}$
  - $u = v + f$, where *u* - value to be returned to the customer
  - $f = \frac{\beta_0(us - \sigma t)}{s(\beta_0 - \beta)}$
  - $\Delta s = -\sigma$
  - $\Delta t = -\frac{\sigma t}{s}$
- In case of deposit:
  - The current share rate must be defined to mint the required number of shares
  - $\Delta t = v$, where *v* - amount of tokens to be deposited
  - $\Delta s = v\lambda$, where λ- share rate as deposit time
  - $\lambda = \frac{s}{\tau}$, where τ - total amount of USD-stable **aligned available**
  - $\tau = t + \frac{I(\beta - \beta_0)}{\beta_0}$, where *I* - total income. So roughly, τ is the total of USD-stable available for customers - deposited amount + income with cleared fee
  - $I = T - v - t - f$, where *T* - total amount of USD-stable **unaligned available**
  - $T = \sum_{i \in E} b_i + \sum_{j \in U} c_j$, where *E* - set of ERC20 pools, *b* - ERC20 balance of the particular pool, *U* - set of ERC4626 pools, *c* - ERC20 balance of the **underlying** token of the particular pool. *It's important to mention that v must be subtracted as it's already supposed to be deposited into the pools (but not to stakers)*

High-level specification update

The interactive update for the high-level specification can be seen here

Low-level specification update

N	Human description	Formal description
		$\forall\, p \in PoolManager,\ o \in bytes32,\ v \in uint256,\ s \in uint256,\ m \in uint256,\ \sigma = msg.sender$
		$\rho = \frac{p.totalDepositedSupply}{p.totalSharesSupply},$
		$T = \sum_{\varsigma \in p.pools20} \varsigma.pool.balanceOf(p.poolKeeper).let\left\{(\varsigma.pool.d > 18)? \frac{it}{10^{\sigma.pool.d-18}} : 10^{18-\varsigma.pool.d} it\right\} +$
		$\sum_{\varsigma \in p.pools4626.} \sigma.pool.pool.convertToAssets(\varsigma.pool.balanceOf(p.poolKeeper))$
		$.let\left\{(\varsigma.pool.d > 18)? \frac{it}{10^{\sigma.pool.d-18}} : 10^{18-\varsigma.pool.d} it\right\},$
		$I(v) = T - v - p.totalDepositsSupply - p.commission,$
		$\tau(v) = p.totalDepositsSupply + \frac{I(v)p.apyFormatter}{APY\_FACTOR}$
		$\Delta d(s) = \frac{sp.totalDepositedSupply}{p.totalSharesAmount}$
		$X(v) = T - p.commission - \frac{(T-p.commission-\Delta d(s))p.apyFormatter}{API\_FACTOR}$

		$\kappa = \frac{(API\_FACTOR - p.apyFormatter)(v \cdot p.totalSharesSupply - s \cdot p.totalDepositedSupply)}{p.apyFormatter \cdot p.totalSharesSupply}$
PDI.1	Only owner can call 'deposit'	$\{p.owner \neq \sigma\}p.deposit(o, v)\{S = false\}$
PDI.2	If the operation is already handled, deposit fails	$\{p.deposits[o]\}p.deposit(o, v)\{S = false\}$
PDI.3	If no value is provided, deposit fails	$\{p.v = 0\}p.deposit(o, v)\{S = false\}$
PDI.4	In case of no total deposit, deposit fails	$\{p.totalDepositedSupply = 0\}p.deposit(o, v)\{S = false\}$
PDI.5	In case of no total shares, deposit fails	$\{p.totalSharesSupply = 0\}p.deposit(o, v)\{S = false\}$
PDI.6	Otherwise, deposit is successful	$\{p.owner = \sigma \land \neg p.deposits[o] \land p.v > 0 \land p.totalDepositedSupply > 0 \land p.totalSharesSupply > 0\}p.deposit(o, v)\{S = true\}$
PWI.1	Only owner can call 'withdraw'	$\{p.owner \neq \sigma\}p.withdraw(o, v, s, m)\{S = false\}$
PWI.2	If the operation is already handled, withdraw fails	$\{p.withdrawals[o]\}p.withdraw(o, v, s, m)\{S = false\}$
PWI.3	If no shares provided, withdraw fails	$\{s = 0\}p.withdraw(o, v, s, m)\{S = false\}$

PWI.4	If no value provided, withdraw fails	$\{v \ = 0\}p.withdraw(o, v, s, m)\{S = false\}$
PWI.5	If amount of shares exceeds total amount of shares, withdraw fails	$\{s > p.totalSharesSupply\}p.withdraw(o, v, s, m)\{S = false\}$
PWI.6	In case of loss, withdraw is not allowed	$\{v < s\rho\}p.withdraw(o, v, s, m)\{S = false\}$
PWI.7	In case of too many shares, withdraw must fail	$\{Tp.totalSharesSupply < p.totalSharesSupply(p.totalDepositedSupply + p.commission) -$ $p.totalDepositedSupply \cdot s\}p.withdraw(o, v, s, m)\{S = false\}$
PWI.7a	In case price is not decreased, withdraw must fail	$\{v(p.totalSharesSupply - s) > sX(v)\}p.withdraw(o, v, s, m)\{S = false\}$
PWI.8	In case of no total deposit, withdraw fails	$\{p.totalDepositedSupply = 0\}p.withdraw(o, v, s, m)\{S = false\}$
PWI.9	In case of no total shares, withdraw fails	$\{p.totalSharesSupply = 0\}p.withdraw(o, v, s, m)\{S = false\}$

PWI.10	Otherwise, withdraw is successful	$\{p.owner = \sigma \land \neg p.withdrawals[o] \land p.totalSharesSupply > 0 \land p.totalDepositedSupply > 0 \land s \leq p.totalSharesSupply \land v \cdot p.totalSharesSupply \geq s \cdot p.totalDepositedSupply$ $\{T \geq (p.totalDepositedSupply + p.commission + \kappa) - \frac{p.totalDepistedSupply \cdot s}{p.totalSharesSupply} \land v(p.totalSharesSupply - s) \leq sX(s) \land v > 0 \land s > 0\}p.withdraw(o, v, s, m)\{S = true\}$
PAI.1	Only owner can call 'acquire'	$\{p.owner \neq \sigma\}p.subtractCommission(v)\{S = false\}$
PAI.2	In case of zero value, acquire must fail	$\{v = 0\}p.subtractCommission(v)\{S = false\}$
PAI.3	If available fees less than required, acquire must fail	$\{v > p.commission\}p.subtractCommission(v)\{S = false\}$
PAI.4	Otherwise, acquire is successful	$\{p.owner = \sigma \land v > 0 \land v \leq p.commission\}p.subtractCommission(v)\{S = true\}$
PDO.1	Total deposit amount is increased by value	$\frac{\{\ \}p.deposit(o,v)\{S=true\}}{\{\forall x{:}x=p.totalDepositedSupply\}p.deposit(o,v)\{p.totalDepositedSupply=x+v\}}$
PDO.2	Deposit operation is handled	$\frac{\{\ \}p.deposit(o,v)\{S=true\}}{\{\ \}p.deposit(o,v)\{p.deposits[o]\}}$
PDO.3	Total shares	$\frac{\{\ \}p.deposit(o,v)\{S=true\}}{\{\forall x{:}x=p.totalSharesSupply\}p.deposit(o,v)\left\{p.totalSharesSupply=x+v\frac{p.totalSharsSupply}{\tau(v)}\right\}}$

	amount is increased by value multiplied by deposit share price	
PWO.1	Withdrawal operation is handled	$\frac{\{\ \}p,withdraw(o,v,s,m)\{S=true\}}{\{\ \}p.withdraw(o,v,s,m)\{p.withdrawals[o]\}}$
PWO.2	Total shares are decreased by the amount provided	$\frac{\{\ \}p,withdraw(o,v,s,m)\{S=true\}}{\{\forall x:x=p.totalSharesAmount\}p.withdraw(o,v,s,m)\{p.totalSharesAmount=x-s\}}$
PWO.3	Total deposit is decreased accordingly	$\frac{\{\ \}p,withdraw(o,v,s,m)\{S=true\}}{\{\forall x,y:x=p.totalDspositedAmount,y=p.totalSharesAmount\}p.withdraw(o,v,s,m)\left\{p.totalDepositedAmount=x-\frac{xs}{y}\right\}}$
PWO.4	Total fee in increased accordingly	$\forall f: f = p.commission,$ $\frac{\{\ \}p,withdraw(o,v,s,m)\{S=true\}}{\{\forall x,y:x=p.totalDspositedAmount,y=p.totalSharesAmount\}p.withdraw(o,v,s,m)\{p.commission=f+\kappa\}}$
PAO.1	Available fees are decreased by the value provided	$\frac{\{\ \}p,subtractCommission(v)\{S=true\}}{\{\forall x:x=p.commission,\}p.subtractCommission(v)\{p.commission=x-v\}}$

Found issues

Smart contract issues

Location	Issue	Status
PoolManager.deposit	No check for zero total supply	Solved by initial investment
PoolManager.deposit	No check for positive value	Solved by oracle
PoolManager.withdraw	No check for positive shares	Solved by oracle
PoolManager.withdraw	No check for positive value	Solved by oracle
PoolManager.withdraw	No check for zero total deposit	Solved by initial investment
PoolManager.withdraw	No check for zero total shares	Solved by initial investment
PoolManager.subtractCommission	No check for zero value	Solved by oracle
PoolManager.subtractCommission	No check if fee is available	Solved by oracle
PoolManager..setAPYFormatter		
PoolManager.addPool20	These methods can ruin the business logic	Solved by agreement with users
PoolManager.setPool20		
PoolManager.popPool20		
PoolManager.addPool4626		
PoolManager.setPool4626		
PoolManager.popPool4626		

Oracle issues

`41`	`Common`	`Common`	`Major`	Race condition is possible	Waived, by agreement with users

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.

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