Molecula Formal Verification Report

Prepared by Pruvendo on 04/09/25

Version 2.1.8

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-public/tree/main/blockchain/solidity/contracts, commit **bc246611ad7c5c0ad7135bd90e0b8763584b8a5a**, made on 03/12/25, where formal verification was performed against the smart contracts of core part and Nitrogen only (Carbon was not considered). While a number of bugs were found, all the major and critical ones were fixed by the development team, while others don't have any practical meaning and don't bring any risk. 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).

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.

The Molecula project has several scenarios for clients (UX-scenarios). Each UX-scenario exploits a sequence of contracts and is initiated by a client. The exploited scenarios are allocated in different contracts and are activated according to the project business logic. The UX-scenarios which serve the same business goal are united together, on the same figure, or in subscenarios.

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
M	Abstraction for message, heterogeneous type, that has the following fields: ● r - Recipient (as A) ● a - Amount(Value) (as N) ● d - Data as an arbitrary nullable type

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 IV 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 IV 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 <u>without</u> apostrophe - before transformation
        - Object <u>with</u> 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 <u>before</u> and <u>after</u> 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 related to either:

- Language-specific behavior, or
- Restrictions from outer systems (such as web3 applications), or
- 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, or
    - transformed from high-level representation into low-level one using mapping discussed above[1]

As above:

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

Stage 7: Conversion to Coq

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

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

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

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

a DSL over Gallina (OCaml dialect used by Coq) that it's syntactically very close to Solidity[3]. So, just see the following example:

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

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

Currently the following translators to Ursus are implemented:
- TVM Solidity (Everscale, GOSH)
- Classic Solidity (Ethereum, Tron, other EVM-compatible chains)
- Rust (MultiversX)
- FunC (TonCoin)

Ursus also can be used as a primary language for development, with later translation to Solidity. Such an approach makes a formal verification easier, as Ursus prevents a developer from introducing some code patterns that bring some complication for the verifications.

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

Evals&Execs

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

[3] Everscale Solidity dialect is used in the present example

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, or
- *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:
    - Specification bug - specification is to be changed, or
    - Minor note - specification is to be changed, while this note must be reflected in the final report, or
    - 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**.

Formal verification

Business-level specification

Introduction

The Molecula project is expected to operate as a prospective mutual fund. All client investments in crypto tokens are pooled into a mutual portfolio. A client obtains a share in the portfolio, formed by Molecula. A share is implemented as a liquid token.The liquid token technology is technology, when a supply of tokens changes each time they are bought or sold. The project distributes the pooled tokens into market tokens. Currently, according to documentation they are TRON, TON and Ethereum tokens[4]. Price changes and volatility of the market tokens (observed as indicating variables) induce rebalance of internal token allocation. All internal operations of the fund are secured in internal smart contracts.

The project provides saving facilities, secured by allocation of savings in a blockchain, along with accounting, and sophisticated internal management. All facilities are secured by smart contracts. Internal management is organized with a special internal token, generated within the system with secured control for minting and burning. A client holding a token receives a yield income reward from the system.

The special feature of the project is a gas-saving approach to reduce client costs for operations.

Currently there are two solutions for clients. A Carbon solution operates both with TRON and Etherium crypto tokens, and Nitrogen, which operates only with Ethereum. Other client-oriented facilities are pending. As outlined above, only the Nitrogen scenario is considered in the present document.

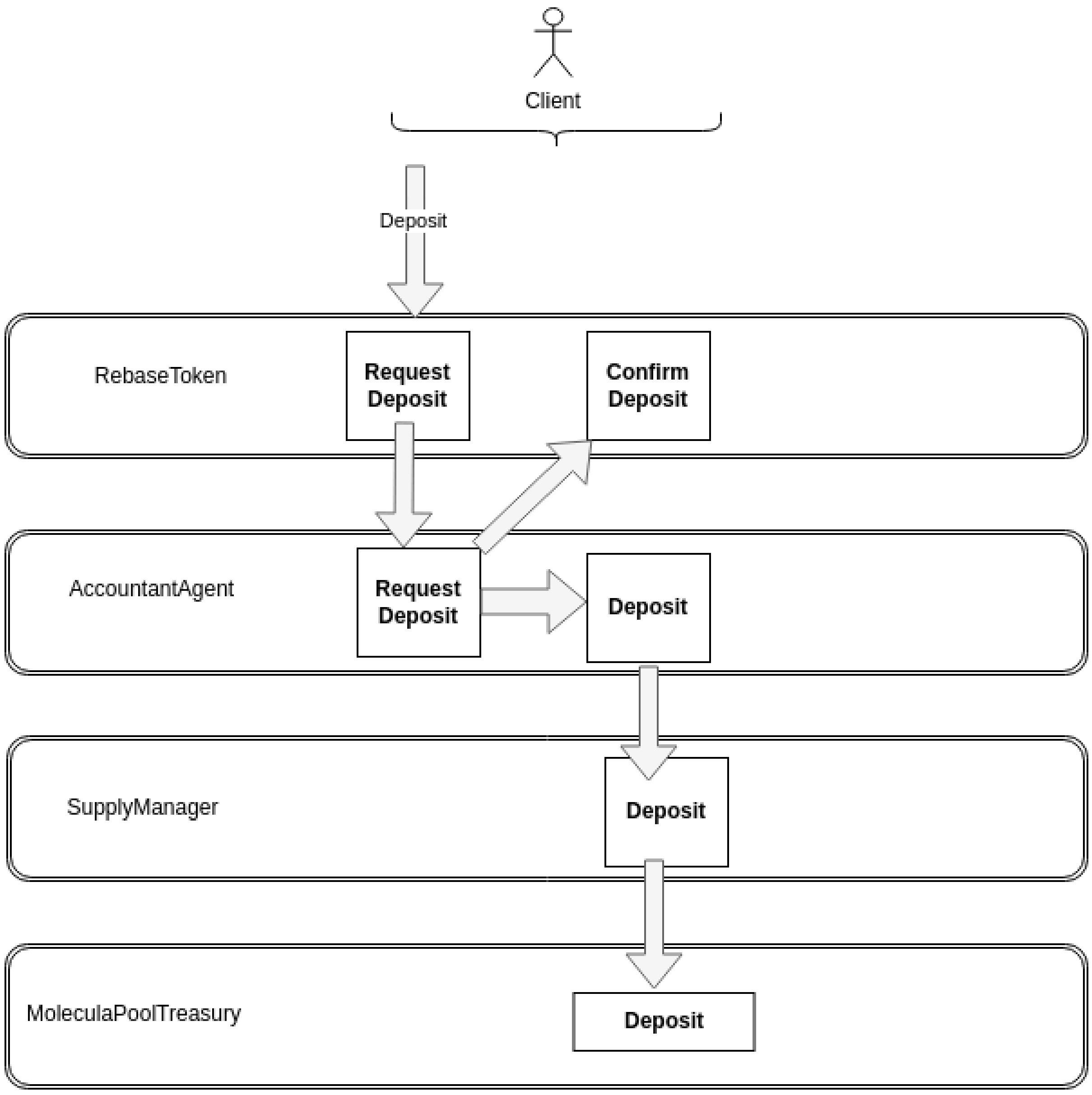
On the top level, for a client, the project contains several main scenarios. They are presented in the figure below. Some of them are initiated by the customers, others by the system itself:

- Initiated by the customers:
    - Deposit
    - Redemption
- Initiated by the system:
    - Yield distribution

[4] Only inter-Ethereum communications are considered throughout the present report

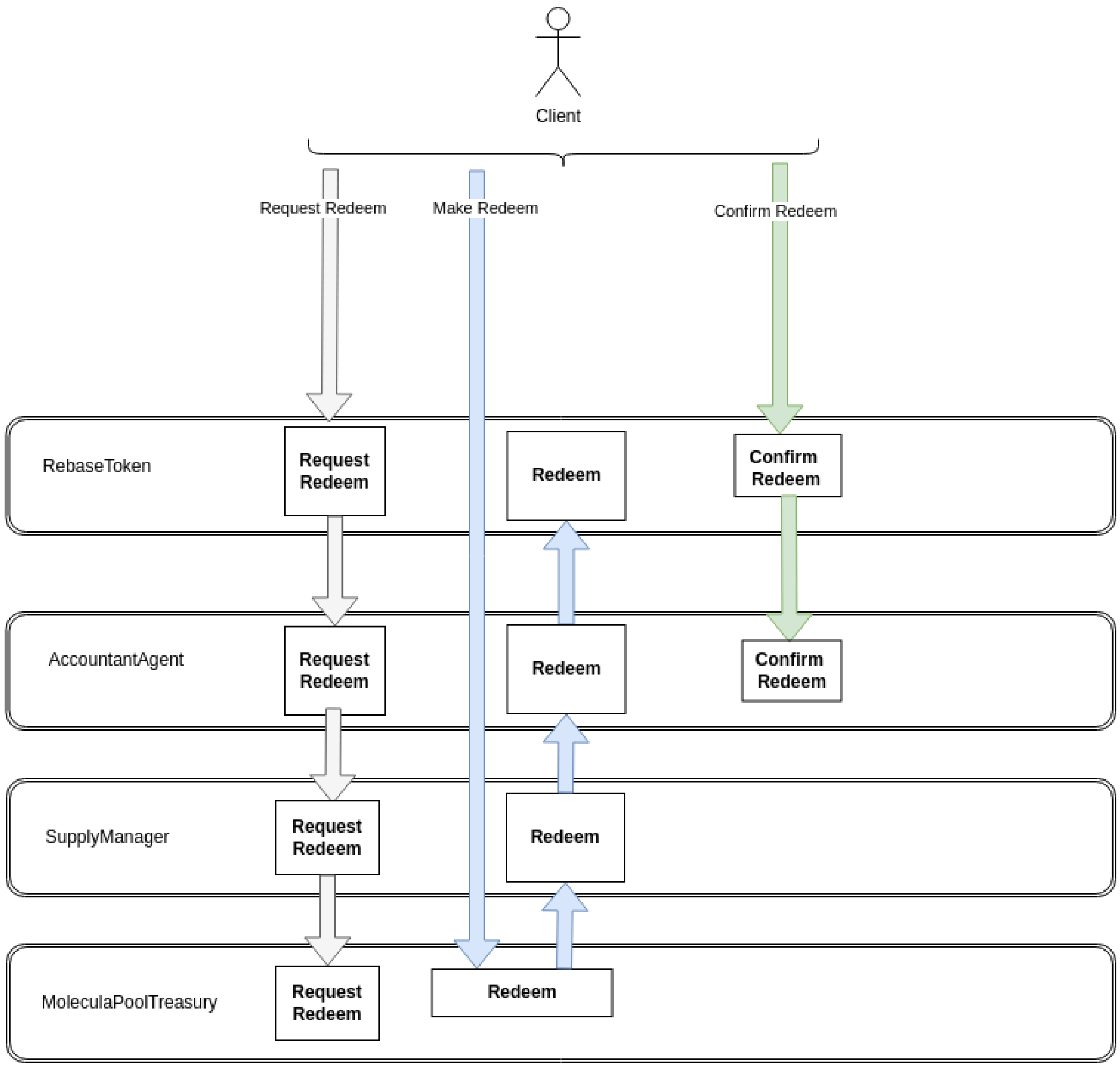
Workflow of the main scenarios

Deposit



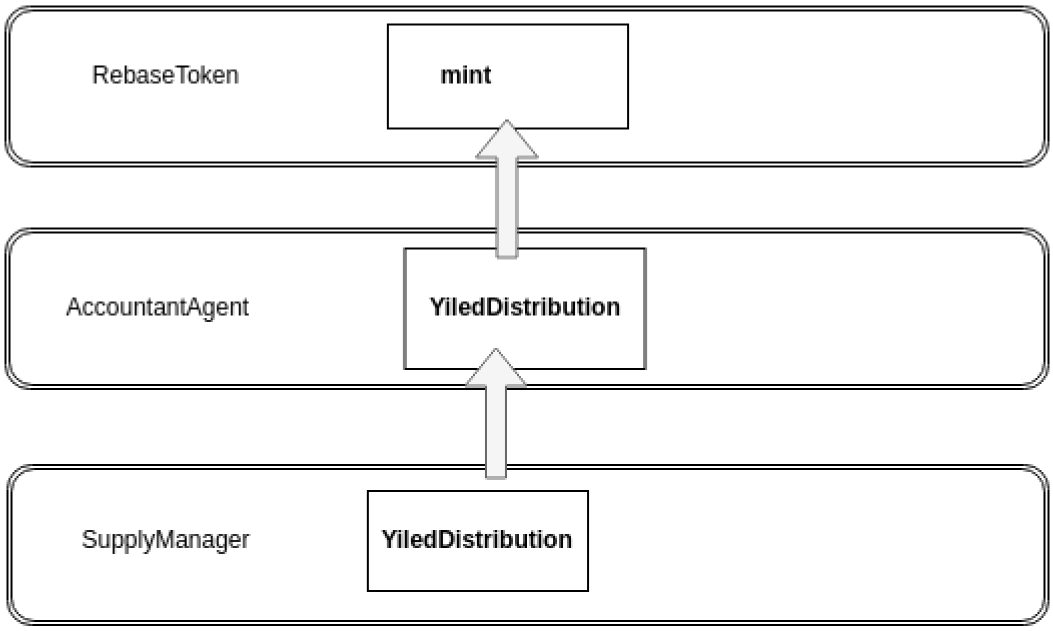
Redemption

Redemption scenarios serve a client to receive savings and income from investments in internal liquid tokens. There are three consecutive scenarios, all are in the following figure.



Yield Distribution

The Yield distribution scenario is periodically invoked by the system, mining the new tokens and distributing them between the system and the investor. The workflow is presented below.



Rebalancing

The goal of the Molecula project is to preserve the value of client's savings in the crypto portfolio of Molecula. The Molecula system allocates assets across various yield-generating protocols to ensure optimal APY returns for the participants. The portfolio consists of stable coins and income tokens, including USDT, USDC, DAI, sDAI, spDAI, USDe, sUSDe, aEthUSDT, aEthUSDC, aEthDAI, sFRAX, FRAX, frxUSD, sFrxUSD, USDS, and sUSDS. The list can be extended.

Molecula crypto portfolio balances potential returns with risk management. Portfolio management within the Molecula project focuses on maintaining an optimal balance between safety and yield. The allocation strategy is based on categorizing DeFi protocols into three classes: Class A (Highest Tier), Class B (Mid Tier), and Class C (Emerging Tier). The classification is determined by the Total Value Locked (TVL) metrics, operational history, and security profile.

Tiers of token classes are

1. Class A (Highest Tier) - the highest degree of reliability:
   - Criteria: Protocols with market TVL above $5 billion and at least 3 years of operation without major security incidents
   - Concentration limit within the tier: No single Class A protocol should exceed 40% of the total portfolio allocation

2. Class B (Mid Tier) - middle degree of reliability:
   - Criteria: Protocols with market TVL above $5 billion or at least 3 years of operation with market TVL above $1 billion
   - Concentration limit within the tier: No single Class B protocol should exceed 20% of the portfolio allocation

3. Class C (Emerging Tier) - the riskiest tokens in the portfolio:
   - Criteria: Newer or smaller protocols with market TVL not less than \$50 million, successful tier-1 security audits, and reputable backing
   - Concentration limit within the tier: combined exposure to Class C protocols should not exceed 10%, with no more than 5% allocated to a single asset

Asset Allocation Guidelines

The Molecula project considers a prudent risk allocation over risk classes.
- 70%-80% in Class A protocols for safety
- 20%-30% in Class B protocols to boost yield
- Up to 10% in Class C protocols for experimental opportunities

The project follows the principle that no token may induce violation of the class concentration limits. This principle is applied if the value of a market token grows and what may misbalance the allowed diversification of risk classes. The principle is also applied if one needs to include a new token into the crypto portfolio.

TVL

The TVL of the portfolio is measured as a sum of values of all tokens in the crypto portfolio. The project differentiates between a current TVL and a target TVL. The current TVL is a metric to measure a realized portfolio perfomance. The current TVL (or a realized TVL) is constructed as follows.
APY_i - a yield (APY) of asset i during a period since the last rebalance of the whole portfolio
R - a risk score assigned to the token based on its volatility, liquidity, performance of other relevant markets. Regulatory changes may have an impact on risk scores too. Details see below.
T - a predefined risk tolerance level for the entire portfolio, a chosen portfolio risk
$W_i = \frac{Y_i/R_i}{\sum_i \frac{Y_i}{R_i}}$ - a risk-weighted APY of a token i

Crypto portfolio management is a maximization problem with a constraint:
Maximize a weighted sum of tokens' APYs

$$TVL_{portfolio} = \sum_i (W_i * Y_i)$$

subject to risk tolerance

$$\Sigma_i (W_i * R_i) < T$$

Rebalancing frequency is not fixed, but can be performed on a regular basis.
The target TVL is a value that the portfolio aims to reach. A comparison of a target and a realized TVL is one of the triggers for portfolio restructuring actions.

Risk score is assigned to every token type in the crypto portfolio. It is a reflection of token volatility and its potential for growth or a loss. Risk scores allow a more developed allocation strategy which reduces the likelihood of overexposing the portfolio to high-risk assets.

- Risk scores range from 1 (the lowest risk) to 100 (the highest risk)
- The risk score for each token is determined based on properties of the token such as token price volatility, token historical performance, liquidity, and regulatory factors

More detailed methodology on how to use market performance to assign risk scores is not available, including dynamic readjustment rules and relevant data sources.

Risk Management

Risk management in the Molecula project is designed to minimize exposure to potential losses while optimizing returns. This involves regular rebalancing of the portfolio based on predefined rules and mathematical models. General Rebalancing Rules of Molecula are

- Rebalancing ensures compliance with class risk concentration limits and adjusts allocations based on yield opportunities
- Tokens which exceed their allowed percentage due to growth sold on the market from the crypto portfolio during rebalancing
- Before rebalancing, price inefficiencies such as slippage or off-peg deviations are assessed to avoid unnecessary losses and transaction costs

Diversification describes risk spread within a class, if multiple options exist within a class. This allows to avoid idiosyncratic risks or risk concentration on a group of tokens within a class. Rebalancing is a tool to implement diversification. It is activated when performance of the crypto portfolio or of an individual token deviates from the target by a certain percentage, when arbitrage opportunities arise, or due to significant price/volatility changes. This aims to keep the Molecula crypto fund overall risk exposure within acceptable limits, balancing potential returns with risk management. Acceptable risk limits for the whole crypto portfolio and how to measure it are unavailable. Rebalancing decisions are made automatically when some criteria are met:

- If a weight of an asset deviates by more than a fixed percentage (e.g., 5%) from the target weight, a rebalance is activated
- The crypto portfolio holds different tokens, for which an arbitrage opportunity may arise. For this case there is an arbitrage rebalance procedure. It is activated when the price difference between any two tokens exceeds 1%

This aims to keep the Molecula crypto fund overall risk exposure within acceptable limits, balancing potential returns with the chosen risk management strategy.

Frequency of Rebalancing

Rebalancing is scheduled either periodically (e.g., monthly or quarterly) or triggered by significant market changes such as yield fluctuations or new opportunities in higher-tier protocols.

An instant liquidity reservation (immediate execution of obligations before clients) is anticipated and is implemented with the redeem operations. The Molecula system makes a stablecoin reserve (5-10%) for liquidity to leave the crypto fund intact.

Assessing Price Inefficiencies

Portfolio rebalancing is always a trade-off between costs and returns. This may cause rebalance inefficiencies, for example, when swapping stable coins incurs slippage or liquidity-related losses. To evaluate the cost-effectiveness of rebalancing the project suggests the following formula:

$Time = \frac{C}{P(Y_2 - Y_1)}$, where:

$Time$ - a break-even time (years)
C - one-time cost of the swap (e.g., gas fees or slippage losses)
P - principal amount being reallocated
Y_1 - current annual yield (APY) of the token bought from the market
Y_2 - annual yield of the token bought from the market
The value of $Y_2 - Y_1$ must be strictly positive

Details

All the addresses must be non-zero, all the values must be non-negative.

ERC4626 Tokens

These ERC20-based tokens are yield-bearing ones that means that:

- They are derived from some original ERC20 tokens
- The original tokens are somehow locked while ERC4626 counterparts exist
- When released, the ERC4626 tokens are burnt (while may be converted into another amount of original tokens than were locked (that means a profit))
- In addition to a regular transfer and allowance, permit (as defined by ERC2612) is also supported
- The rest of features can be learnt by reading the ERC4626 standard

Common Scenarios

Supply Manager - specific

The *SupplyManager* contract manages pool staking, making and withdrawing deposits along with distributing yield in the Molecula project.

Init

The scenario is intended to initialize the *SupplyManager* contract,
The contract constructor takes the following parameters:

- An owner of the contract, responsible for governance (must be non-zero)
- An entity responsible for distributing yield

- An address of the *MoleculaPool* (here and later, *MoleculaPool* term refers both to the old *MoleculaPool* contract as well as to the actual *MoleculaPoolTreasury*) contract
- The owner's share used for revenue distribution
- The guardian (must be non-zero)

The contract sets up:

- A reference to the *MoleculaPool* contract
- V_d - total deposit supply, a state variable, V_d is considered as always positive being an axiom
- *S* - a total shares supply
- Control variables responsible for shares APY for recapitalization and rewards
- AYD - an address of the yield distributor

MoleculaPoolTreasury Total Supply

The *totalSupply* scenario provides a formatted version of the total value deposited in the *MoleculaPoolTreasury*.

Input - no input

Output - an adjusted total supply of the Molecula Pool, if necessary with values deposited for redeem:

- $v(t, x)$- value converted to assets, if applicable, for the particular token
  - $t \in ERC20 \Rightarrow v(t, x) = x$
  - $t \in ERC4626 \Rightarrow v(t, x) = t.convertToAssets(x)$
- $s(t)$ - total supply of the particular token
- $r(t)$ - value to redeem for the particular token
- $n(t)$ - number of decimals for each particular token
- Then , the output is $\sum_{t \in Tokens} 10^{18-n(t)}(s(t) - r(t))$

The scenario returns the adjusted total supply (*V*).

SupplyManager Total Supply

This scenario provides the total value in the Molecula pool excluding the amount dedicated to Molecula beneficiaries.

Input - no input

Output - total value deposited in the molecula pool but the System share

Set Agent

The *setAgent* scenario is used to authorize or deauthorize an *Agent* in the system by a presence in the approved list or not.

Input - an address of the Agent being authorized or unauthorized.
Output - no output
Workflow

- Each agent is presented in a list of agents only once
- Each agent has either a true or false label
- Can be labeled false only if earlier was true
- Can be labeled true only if earlier was false or was not present

Distribute Yield

The *Distribute Yield* scenario distributes accrued income (or extra yield) generated by the *MoleculaPoolTreasury* among authorized parties. It validates inputs, calculates the extra yield, distributes it, including a recapitalization and reports a result.

Input:

- A list of agents and their shares
- A new share of the system income

Requirements:

- All the agents are known
- No duplicated agents
- The overall share of all the provided agents is 100%

Output : None

Workflow :

- *Real Total Supply* - total supply, kept by *MoleculaPoolTreasury*
- No action from the scenario in case of no income
- *Real Yield* is the income (difference between the *Real Total Supply* and *Documented Total Supply*)
- *Extra Yield* is a System share
- *Current Yield* is an investor's share
- the *New Total Supply* is the *Real Total Supply*, but the System share
- *Shares to mint* is an *Extra Yield* normalized to shares
- *Shares to distribute* is a *Shares To Mint* added to the amount of the previously locked shares
- Each party gets a number of shares proportional to their deposit
- New System share is assigned
- New *Documented Total Supply* is increased by *Real Yield*
- No locked shares any more

Set AYD

The scenario Set AYD updates a yield distributing address.

Requirements:

- The sender is the owner

Input - The new address to be set as the authorized yield distributor

Output - None

Workflow:

- Update the State Variable

MoleculaPoolTreasury - specific

The *MoleculaPoolTreasury* contract is designed to manage pools of tokens (both *ERC20* and *ERC4626*) and facilitate operations such as deposits, redemptions, and total supply calculations. It operates with values (of tokens), in contrast to the *SupplyManager*, which operates with quantities (of tokens). The contract provides a structure for organizing token pools, normalizing their balances, and handling interactions with *SupplyManager*. A whitelist mechanism restricts which addresses can interact with certain functions.

Init

Intended to initialize the contract *MoleculaPoolTreasury*

Inputs

- An address that will own the contract and have administrative privileges
- An address that is permitted to redeem tokens
- An array of tokens (both ERC20), including their normalization factor
- An address responsible for maintaining the pool
- An address for token supply operations management
- A guardian address

Output - None

Workflow

- Validate Input Addresses that they are non-zero
- Assign role-based addresses for the *Owner*, *authorizedRedeemer*, *poolKeeper*, and *SupplyManager*
- Initialize *token pool* and prevent duplicate tokens
- *SupplyManager* becomes immutable

Global notes

- A token cannot be added to the pool more than once
- A pool can only be removed if its balance is zero and value to redeem is zero
- Only designated roles can execute critical functions
- Decimals of the tokens are calculated automatically, assuming the support *IERC20Metadata* interface.Tokens that do not claim this interface, are unsupported

Token Operations

Represents a number of operations related to the list of tokens assigned to the *MoleculaPoolTreasury*.

<u>removeToken</u>

The operation is intended to remove a token from the set of handled tokens.

Requirements:

- Can be called by the Owner only
- Token must exist
- Token must have zero balance
- Token must have zero value to redeem

Input - Token to be removed.
Output - None.

<u>addToken</u>

The operation is intended to add a token to the set of handled tokens.

Requirements:

- Can be called by the Owner only
- Token must not exist in the pool
- Decimals of the tokens are calculated automatically, assuming the support *IERC20Metadata* interface.Tokens that do not claim this interface, are unsupported

Input - Token to be added.
Output - None.

Execute

This scenario allows an execution of transactions on behalf of a contract from the whitelist.

Requirements:

- Only the Pool Keeper can call this function.
- The execution must not be paused
- If this is *approve* function:
    - the token must be in the token pool
    - the token must not be blocked
    - The spender must be whitelisted
- Otherwise:
    - The target must be whitelisted

Input:

- A target address

- Encoded function call data containing the function selector and parameters.

Output:
- Returned data from the executed function call.

Migrate

The *migrate* scenario transfers state and assets from an old *MoleculaPool* contract (*MoleculaPool*) to the new one (*MoleculaPoolTreasury*). This ensures that liquidity, token pools, and agent permissions are correctly migrated while maintaining continuity of operations.

Requirements:
- Can be initiated by the *SupplyManager* only, that, in their turn, can be called exclusively by the *Owner*

Inputs:
- An address of an old MoleculaPool contract

Outputs:
- An update of the new contract's states

Workflow:
- All existing token pool in the old *MoleculaPool* are either matched or added in the new pool
- The full balance of each token in old pools is transferred from the old *PoolKeeper* to the new pool.
- All authorized agents from the old pool remain authorized in the new pool
- Decimals of the tokens are calculated automatically, assuming the support *IERC20Metadata* interface.Tokens that do not claim this interface, are unsupported
- High redeem values are rejected, while low redeem values are simply dropped

Set Agent

This scenario *Set Agent* defines a mechanism for authorizing or deauthorizing an *Agent* to interact only with an *ERC20* or *ERC4626* token on behalf of a pool.

Requirements:
- The scenario must be initiated by *SupplyManager* only

Input:
- The Ethereum address of the Agent to be authorized or unauthorized.
- A flag indicating whether to authorize (true) or deauthorize (false) the *Agent*

Output:
- The token approval state for the Agent is altered

Notes:

- An Agent is either fully authorized or completely restricted

Set Whitelist

The scenario Whitelist manages a whitelist of addresses. It allows only the contract owner to add or remove addresses from the whitelist.

Requirements:
- Only the contract owner can call *addInWhiteList* and *deleteFromWhiteList*
- An address cannot be added if it is already in the whitelist
- An address cannot be removed if it is not in the whitelist

Input:
- An ethereum address to be added to or removed from the whitelist

Output:
- The whitelist mapping is to be modified

Guardian operations

This scenario represents different guardian-related operations:

- Guardian change
  - Can be invoked by the owner only
  - Sets a new guardian
- All pauses
  - Can be invoked either by the guardian or by the owner
  - Pause Execute
  - Pause Redeem
  - Pause All
- All unpauses
  - Can be invoked by the owner only
  - Unpause Execute
  - Unpause Redeem
  - Unpause all
  - Block or unblock the specific token

Token - specific

The *RebaseToken* implements a paradigm of a liquid token, where:
- Its value is always stable
- Each owner owns some fixed amount of shares
- The rate of shares is flexible

Mint

The mint function allows the contract owner to mint a specified number of shares (that cost to the value provided) and assign them to a given receiver.

Requirements:

- The scenario is initiated by the *SupplyManager* only
- The token is non-zero

Input:

- A volume to be minted.
- An Ethereum address that will receive the minted shares

Output:

- A number of shares that were successfully minted.
- After minting, the receiver's balance should increase by a minted quantity.

Workflow:

- The scenario returns a number of minted shares

Transfer

The scenario allows a token holder to send a specified amount of shares to another address.

Requirements:

- The sender must have enough shares to complete the transfer.
- The recipient address must be valid (not zero)

Input:

- The recipient's Ethereum address
- A volume to be transferred.

Output

- A boolean value indicating whether the transfer was successful

Transfer From

The transferFrom scenario allows a third party (*spender*) to transfer a specified amount of shares (indicated by its value) from one address (*from*) to another (*to*) on behalf of the sender. It ensures that the spender has sufficient allowance before executing the transfer.

Requirements:

- A spender (*caller*) must have sufficient allowance to transfer shares on behalf of *from*
- A sender (*from*) must have enough shares to complete the transfer
- A recipient address (*to*) must be valid

Input:

- An address from which the shares will be deducted
- A recipient's Ethereum address
- A volume to be transferred

Output:

- A boolean value indicating whether the transfer was successful (true)
- A correct reduction of the spender's allowance
- Correct balance update

Approve

The *approve* function allows the caller (*msg.sender*) to set an allowance for a spender, granting them permission to spend up to a specified number of shares on their behalf. This function is commonly used in *ERC20* token contracts for delegated transfers.

Requirements:

- Always true

Input:

- An address with a permission to spend the caller’s tokens
- A maximum amount of shares the spender can spend

Output:

- Always true
- In case of limited allowance, the allowance corresponds to the number provided
- In case of unlimited allowance, the allowance corresponds to the number provided[5]

Setters

Set the values of the common parameters

Requirements:

- All the setters must be initiated by the *Owner*
- Minimal values must be respected

Input:

- Value to be set

Output:

- The value is to be assigned to the corresponding State variable

Old Molecula Pool - specific

While these scenarios are considered in HLS and LLS, they are deprecated and skipped in the current section!!!

Deposit

Token deposit request

Initiates the deposit operation.

[5] All the cases must be wrapped by the separate statements

Requirements:
- Only the deposit owner or its operator can initiate this operations
- The deposit amount must exceed the minimally allowed size

Input:
- deposit amount
- deposit owner
- deposit beneficiary

Output:
- Request id

Workflow:
- New deposit request is created in *Pending* state
- Request is sent to the *Accountant* for further handling

Nitrogen deposit

Accepts deposit from the deposit owner.

Requirements:
- Can be called by the *RebaseToken* only
- Must have zero value
- Request Deposit must not be paused

Input:
- Request id
- Deposit owner
- Deposit value

Output - None

Workflow:
- Assets are transferred from the deposit owner to the *Accountant*
- Approve for the whole amount is granted to the *MoleculaPooTreasury*
- Request is transferred to the *SupplyManager*
- Request is transferred back to the *RebaseToken* with confirmation

SupplyManager Deposit

Updates amounts of shares and deposited values.

Requirements:
- Only *Agent* can call this scenario

Input:

- Token
- Request Id
- Deposit value

Output - None

Workflow:

- State variables updates

V_d a current total value of client deposits, submitted earlier to the Molecula.

S - an instant total number of internal shares in Molecula

V - a current value of instant total number of internal shares in Molecula

A client supplies a value v, which with a price $p = \frac{S}{V}$ is converted to

$s = v\frac{S}{V}$ units of shares in Molecula.

Update of state variables

$S += s$

$V_d += v$

- Request is forwarded to the *MoleculaPoolTreasury*

MoleculaPoolTreasury Deposit

The deposit scenario allows making deposits of ERC-20 token standard into the system after a normalization of their values and their transfer into the contract. It supports two types of token pools:

- *ERC-20* tokens: Normalizes the deposit amount directly
- *ERC-4626* tokens: Converts the deposited amount into asset shares before normalization

Requirements:

- Only the *SupplyManager* can call this function
- The token must exist in the token pool
- Token transfers must be successful or the request is rejected

Input

- The ERC20/ERC4626 token to deposit
- A request ID associated with the operation
- An address from which the tokens are being transferred
- An amount of tokens being deposited

Output

- A normalized deposit amount after conversion, adjusted according to the token pool
    - If the token is ERC20, normalize its value directly
    - If the token is ERC4626, convert it, then normalize

Token Deposit Confirm

Completes deposit request and mints the required amount of shares in favor of the deposit beneficiary.

Requirements:
- Only *Accountant* can initiate this scenario
- Deposit request must be in *Pending* state

Input:
- Request Id
- Shares to mint

Output - None

Redeem

Request Redeem

Token Request Redeem

Burns the user shares and initiates the redeem operation

Requirements:
- Can be initiated by the owners of the shares or its operator
- The amount to redeem must exceed the minimal value

Input:
- Amount of shares to redeem
- Owner of the shares
- Redeem beneficiary

Output - request ID

Workflow:
- Request generated
- Shares burned
- Request is forwarded to the *Accountant*

Nitrogen Request Redeem

Just forwards the request to the *SupplyManager Request Redeem*.

Requirements:
- Can be initiated by *RebaseToken* only
- Must have zero value
- Request Redeem must not be paused

Input:

- Request Id
- Amount of shares to redeem

Output - nothing

SupplyManager Redeem Request

Recalculates all the state variables to reflect the changed amount of shares as well as changed total supply.

Requirements:

- Can be initiated by *Accountant* only
- Must have shares to redeem
- Request must be in None state

Input:

- Token
- Request Id
- Amount of shares to redeem

Output - value to redeem

Workflow:

- State variables updated

V_d - a current total value of client deposits, submitted earlier to the Molecula.

S - an instant total number of internal shares in Molecula

V - a current value of the instant total number of internal shares in Molecula

A client supplies a desired to withdraw number of tokens s, which with a price $p = \frac{S}{V}$ is converted to an equivalent value $v = s\frac{V}{S}$.

If a value of internal tokens V exceeds a value of client deposited tokens V_d, then a client receives an accrued revenue. The withdrawal process is adjusted for that case.

$y = s(V - V_d)/S$ - total accumulated value from an accrued income in the system for the claimed token number s. This value must be reserved in system shares. Let

$y_b = y_b + s\frac{V-V_d}{V}$ be a number of blocked shares equivalent to the accumulated value

- Request is forwarded into *MoleculaPoolTreasury*
- Request is moved into Pending state

MoleculaPoolTreasury Redeem Request

This scenario manages client requests to get tokens and rewards from savings in the token *pool*. It updates the redemption balance of each token (*valueToRedeem*) and returns the

amount formatted in the correct token decimals, ensuring compatibility for future transferFrom calls.
Requirements:

- Only the *SupplyManager* can call this scenario
- The token must exist in the token pool

Input:

- The *ERC20* or *ERC4626* token address for redemption
- An amount to redeem

Output: Token amount, adjusted to the token's native decimals

Workflow:

- The function updates *valueToRedeem* for the token, ensuring that total supply calculations remain accurate after redemptions
- The value is normalized from 18 mUSD decimals to the token's actual decimals
- For *ERC4626* tokens, the function converts assets into shares before returning the value

Redeem

MoleculaPoolTreasury Redeem

This scenario processes token redemptions based on a list of request IDs. It calls the redeem method of the *SupplyManager* contract, retrieves the redeemed *ERC20* token and amount, normalizes the value, and updates the redemption balance (*valueToRedeem*).

Requirements;

- The requestIds array must not be empty
- The redeem operation must not be paused
- No token can be blocked

Input - An array of request IDs that identify the redemption requests.
Output - None;

Workflow

- Executes redemption with *SupplyManager*
- Normalizes the redeemed value
- Updates *valueToRedeem* for each token

SupplyManager Redeem

The Redeem scenario serves to redeem requests, accumulated at the current moment from the same client. A client can make several redeem requests before finalizing a withdrawal with one redeem operation. At the moment there is no way to make a withdrawal from more than from one pool.

A total value for redeem from the pool is $V_{total} = sum_{\{i \in [0, len(A)\}} v[i]$.

Requirements:

- All indentifies $rID \neq 0$, and all requests have correct execution status $\forall i \in [0, A]$: $requestIds[i]].status = Pending$.
- All requests came from the same client, $\forall i, j \in [0, len(A)]$ are two agents, with withdrawal requests, then $i = j$. A value v owned by a client i is positive,i, $v[i] > 0$. **NB!** it must be greater than a gas cost after the Molecula fee to withdraw it.

Input:

- An address of a client
- A list of client's requests $[v[0],..., v[L_{id}]$, $L_{id} > 0$

Output:

- Address of the ERC20 token associated with a client $t \in Address$
- total value redeemed $V_{total} = \sum_{t \in T} v_t[i]$
- Check validity of a request to perform redeem L_{id}.
- Construct a total value $V_{total} = \sum_{v_{t \in L_{val}}} v_t$ to redeem
- Return L_{val} and V_{total}.

Workflow:

- There are no changes in state variables, $S_{before} = S_{after}$, and $V = V_{after}$, $\frac{S_{before}}{V_{before}} = \frac{S_{aftet}}{V_{after}}$
- The scenario modifies only the status and values of the provided collection of withdrawal requisition from the same tID without altering unrelated state variables

Nitrogen Redeem

Transfers amount to redeem from the *MoleculaPoolTreasury* to *Accountant* and forwards the redeeming to the *RebaseToken*.

Requirements:

- The scenario can be initiated by the *SupplyManager* only
- The message must have zero value

Input:

- *MoleculaPoolTreasury* address
- list of request IDs
- list of values to redeem

- total value to redeem

Output - none

RebaseToken Redeem

Moves all the redeemed requests into *Unconfirmed* and returns their total value.

Requirements:

- The scenario can be initiated by *Accountant* only

Input:

- list of requests IDs
- list of redeem values

Output - total value of all the handled requests. The handled request is the request, that:

- exists in the list
- initially located in the *Pending* state

Workflow:

- All the requests in the *Pending* state (as well as in the list):
    - Moved into *Unconfirmed* state
    - Get a value assigned that corresponds to the value provided
- Total value calculated

Redeem Confirm

Token Redeem Confirm

Moves the request into the *Confirmed* state and forwards the confirmation request to the *Accountant*.

Requirements:

- Can be initiated by anybody
- The request must be in *Unconfirmed* state

Input:

- Request Id

Output - None

Nitrogen Redeem Confirm

Transfers the assets to the redeem beneficiary.

Requirements:

- Can be initiated by *RebaseToken* only

Input:
- Beneficiary
- Amount to send

Output - None

Distribute

Distributes the yield by minting the tokens in favor of beneficiaries.

Requirements:
- Can be initiated by *SupplyManager* only
- Zero value is required

Input:
- List of users (beneficiaries)
- List of shares to mint

Output - none

Nitrogen Guardian

This scenario represents different guardian-related operations:

- Guardian change
  - Can be invoked by the owner only
  - Sets a new guardian
- All pauses
  - Can be invoked either by the guardian or by the owner
  - Pause Deposit Request
  - Pause Redeem Request
  - Pause All
- All unpauses
  - Can be invoked by the owner only
  - Unpause Deposit Request
  - Unpause Redeem Request
  - Unpause All

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

As was defined in the previous section, the following scenarios and subscenarios were identified:

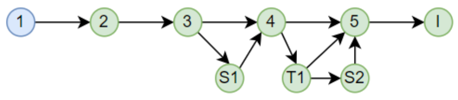
1. Common
    a. Supply Manager - specific
        i. Init
            1. MP2 Total Supply
        ii. Total Supply
        iii. Set Agent
        iv. Distribute Yield
        v. Set AYD
    b. Molecula Pool - specific
        i. Init
        ii. Pool Operations
        iii. Execute
        iv. Migrate
        v. Set Agent
        vi. Set Whitelist
        vii. Guardian
    c. Token - specific
        i. Mint
        ii. Transfer
        iii. Transfer from
        iv. Approve
        v. Setters
    d. Old Molecula Pool - specific
        i. Deposit
        ii. Init
        iii. Pool Operations
        iv. Redeem
        v. Request Redeem
        vi. Set Redeemer
        vii. Set Supply Manager
        viii. Total Supply
2. Deposit
    a. Token Deposit Request
        i. Nitrogen Deposit
            1. SM Deposit
                a. MP Deposit
            2. Token Deposit Confirm
3. Redeem
    a. Request Redeem
        i. Nitrogen Request Redeem
            1. SM Redeem Request

                a. MP Redeem Request
    b. Redeem
        i. MP Redeem
            1. SM Redeem
        ii. Token Redeem
    c. Redeem Confirm
        i. Nitrogen Redeem Confirm
4. Nitrogen-specific
    a. Distribute
    b. Guardian

The details about each scenario can be found in the interactive high-level specification.

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



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

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

Low-level specification

Core part

Supply Manager

Scenario Init

N	Human description	Formal description
		$\forall \sigma \in SupplyManager, o \in A, d \in A, p \in A, f \in \aleph$
		$i = 'constructor', \tau = p.totalSupply(), \tau > 0$
SII.1	The scenario is successful if and only if MoleculaPool.totalSupply is successful and does not return 0	$\frac{\{\forall b \in B\}\tau\{S=b\}}{\{o \neq null \wedge d \neq null \wedge p \neq null \wedge f \leq APY\_FACTOR\}\sigma.i(o,d,p,f)\{S=b\}}$
	In case of success:	
SIO.1	Owner is assigned accordingly	$\frac{\{\}\sigma.i(o,d,p,f)\{S=true\}}{\{\}\sigma.i(o,d,p,f)\{\sigma.\_owner=o\}}$
SIO.2	APY factor is assigned accordingly	$\frac{\{\}\sigma.i(o,d,p,f)\{S=true\}}{\{\}\sigma.i(o,d,p,f)\{\sigma.apyFormatter=f\}}$
SIO.3	AYD is assigned accordingly	$\frac{\{\}\sigma.i(o,d,p,f)\{S=true\}}{\{\}\sigma.i(o,d,p,f)\{\sigma.authorizedYieldDistributor=d\}}$

SIO.4	Molecula Pool is assigned accordingly	$\frac{\{\}\sigma.i(o,d,p,f)\{S=true\}}{\{\}\sigma.i(o,d,p,f)\{\sigma.MOLECULA\_POOL=p\}}$
SIO.5	Total deposited value is a total supply	$\frac{\{\}\sigma.i(o,d,p,f)\{S=true\}}{\{\}\sigma.i(o,d,p,f)\{\sigma.totalDepositedSupply=\tau\}}$
SIO.6	Shares initially distributed using 1:1 rate to the tokens	$\frac{\{\}\sigma.i(o,d,p,f)\{S=true\}}{\{\}\sigma.i(o,d,p,f)\{\sigma.totalSharesSupply=\tau\}}$

Scenario Total Supply

N	Human description	Formal description
		$\forall\sigma \in SupplyManager, \hat{\tau} = \sigma.MOLECULA\_POOL.totalSupply(),$
		$r \in \aleph, \tau = \sigma.totalDepositedSupply, \bar{\tau}: () \rightarrow \aleph$
		$\tau \geq \hat{\tau} \Rightarrow \bar{\tau} = \hat{\tau}$
		$\tau < \hat{\tau} \Rightarrow \bar{\tau} = \tau + \frac{(\hat{\tau}-\tau)\sigma.apyFormatter}{API\_FACTOR}$
STI.1	The scenario is successful if and only if, the Molecula.totalSupply is successful	$\frac{\{\forall b \in B\}\sigma.totalSupply()\{S=b\}}{\{\}\bar{\tau}\{S=b\}}$
STO.1	In case of success, total supply is calculated accordingly	$\frac{\{\}\sigma.totalSupply()\{S=true\}}{\sigma.totalSupply=\bar{\tau}}$

Scenario MP Total Supply

N	Human description	Formal description
		$\forall p \in MoleculaPool, t = p.totalSupply$
		$\tau = \tau_{20} + \tau_{4626} - p.valueToRedeem$
		$\tau_{20} = \sum_{\pi \in p.pools20} 10^{\pi.n} \pi.pool.balanceOf(p.poolKeeper)$
		$\tau_{4626} = \sum_{\pi \in p.pools4626} 10^{\pi.n} \pi.pool.convertToAssets(\pi.pool.balanceOf(p.poolKeeper))$
PTI.1	The scenario is always successful	$\{\}t()\{S = true\}$
PTO.1	In case of success, total supply is calculated accordingly	$t() = \tau$

Scenario Set Agent

N	Human description	Formal description
		$\forall \sigma \in SupplyManager, a \in A, i \in B, \alpha = 'setAgent'$
		$\forall i, j \in [0..\sigma.agentsArray.length): i \neq j \Rightarrow \sigma.agentsArray[i] \neq \sigma.agentsArray[j]$

SAI.1	The scenario is successful if and only if the sender is the owner and the underlying call is successful	$\frac{\{\}\sigma.moleculaPool.setAgent(a,i)\{S=true\}}{\{\}\alpha(\sigma,a,i)\{S=(msg.sender=\sigma.\_owner \land \sigma.agents[a]\neq i)\}}$
	In case of success:	
SAO.1	The agent’s authorization status is updated to the requested value	$\frac{\{\}\alpha(\sigma,a,i)\{S=true\}}{\{\}\alpha(\sigma,a,i)\{\sigma.agents[a]=i\}}$
SAO.1.1		$\frac{\{\}\alpha(\sigma,a,i)\{S=true\}}{\{\}\alpha(\sigma,a,true)\{\sigma.agentsArray[length-1]=a\}}$
SAO.1.2		$\{\}\alpha(\sigma, a, false)\{a \notin \sigma.agentsArray\}$
SAO.1.3		$\frac{\{\}\alpha(\sigma,a,i)\{S=true\}}{\{\}\alpha(\sigma,a,i)\{\uparrow\sigma.moleculaPool.setAgent(a,i)\}}$

Scenario Distribute Yield

N	Human description	Formal description
		$\forall\sigma \in SupplyManager, p \in Parties, f \in \aleph$
		$\lambda = \sigma.distributeYield,$
		$\nu = \sigma.\_validateParties(p)$
		$v(x) = v_a(x) \land v_\sigma(x) \land v_0(x)$

		$v_a(x) = (\forall\pi \in x: \pi.agent \in \sigma.agents) \wedge (\forall\pi_1, \pi_2 \in x: \pi_1 \neq \pi_2 \Rightarrow \pi_1.agent \neq \pi_2.agent)$
		$v_\sigma(x) = \left(\left(\sum_{\pi\in x}\sum_{\rho\in\pi.parties} \rho.portion\right) = FULL\_PORTION\right)$
		$v_0(x) = x.length > 0$
		$y_r = \sigma.MOLECULA\_POOL.totalSupply() - \sigma.totalDepositedSupply$
		$y_c = \frac{y_r \sigma.apyFormatter}{APY\_FACTOR}$
		$y_e = y_r - y_c$
		$\hat{\nu} = \sigma.totalDepositedSupply + y_c$
		$\mu = \frac{y_e \sigma.totalSharesSupply}{\hat{\nu}}$
		$\delta = \mu + \sigma.lockedYieldShares$
SDI.1	The sender must be AYD, otherwise the scenario fails	$\{msg.sender \neq \sigma.authorizedYieldDistributor\}\lambda(p, f)\{S = false\}$
SDI.2	In case of too big APY factor, the scenario fails	$\{f > APY\_FACTOR\}\lambda(p, f)\{S = false\}$
SDI.3	In case of incorrect parties, the scenario fails	$\{\}\nu\{S = false\} \Rightarrow \{\}\lambda(p, f)\{S = false\}$

SDI.3.1		$\{\}\nu\{S = v(p)\}$
SDI.4	Otherwise, the scenario must be successful	$\dfrac{\forall\pi\in p:(\{\}\pi.i.distribute\{\pi.ethValue\}(\pi.party,\frac{\delta\pi.portion}{FULL\_PORTION})\{S=true\})}{msg.sender=\sigma.authorizedYieldDistributor\wedge f\leq APY\_FACTOR\wedge\sigma.realTotalSupply>\sigma.totalDepositedSupply\wedge v(p)\}\lambda(p,f)\{S=true\}}$
	In case of success:	
SDO.1	Agents are instructed to distribute yield	$\dfrac{\{\}\lambda(p,f)\{S=true\}}{\{\}\lambda(p,f)\{\forall\pi\in p:\uparrow\pi.i.distribute\{\pi.ethValue\}(\pi.party,\frac{\delta\pi.portion}{FULL\_PORTION})\}}$
SDO.2	Total shares are increased by minted amount	$\dfrac{\{\}\lambda(p,f)\{S=true\}}{\{\forall x:x=\sigma.totalSharesSupply\}\lambda(p,f)\{\sigma.totalSharesSupply=x+\mu\}}$
SDO.3	Total deposit suply becomes equals to the total supply from Molecula Pool	$\dfrac{\{\}\lambda(p,f)\{S=true\}}{\{\}\lambda(p,f)\{\sigma.totalDepositedSupply=\sigma.MOLECULA\_POOL.totalSupply()\}}$
SDO.4	Locked values are reset	$\dfrac{\{\}\lambda(p,f)\{S=true\}}{\{\}\lambda(p,f)\{\sigma.lockedYieldShares=0\}}$
SDO.5	apyFactor is updated	$\dfrac{\{\}\lambda(p,f)\{S=true\}}{\{\}\lambda(p,f)\{\sigma.apyFormatter=f\}}$

Scenario Set AYD

N	Human description	Formal description
		$\forall\sigma \in SupplyManager, n \in A, d = \sigma.'setAuthorizedYieldDistributor'$
SYI.1	Successful if and only if the sender is the owner and the new address is not the	$\{\}d(n)\{S = (msg.sender = \sigma.\_owner \wedge n \neq null)\}$

	zero address	
	In case of success:	
SYO.1	New AYD is assigned	$\frac{\{\}d(n)\{S=true\}}{\{\}d(n)\{\sigma.authorizedYieldDistributor=n\}}$

Molecula Pool

Scenario Init

N	Human description	Formal description
		$\forall p \in MoleculaPool, o \in A, r \in A, p_{20} \in [Pool], p_{4626} \in [Pool], k \in A, \sigma \in A$
		$i = p.constructor$
		$\kappa_{20} = \forall \pi_1, \pi_2 \in p_{20}: \pi_1 \neq \pi_2 \Rightarrow \pi_1.pool \neq \pi_2.pool$
		$\kappa_{4626} = \forall \pi_1, \pi_2 \in p_{4626}: \pi_1 \neq \pi_2 \Rightarrow \pi_1.pool \neq \pi_2.pool$
		$\kappa = \kappa_{20} \wedge \kappa_{4626}$
PII.1	Scenario is always successful	$\{\}i(o, r, p_{20}, p_{4626}, k, \sigma)\{S = (o \neq 0 \wedge r \neq 0 \wedge k \neq 0 \wedge \sigma \neq 0 \wedge \kappa\}$
PIO.1	Owner is assigned properly	$\{\}i(o, r, p_{20}, p_{4626}, k, \sigma)\{p.\_owner = o\}$
PIO.2	Redeemer is assigned	$\{\}i(o, r, p_{20}, p_{4626}, k, \sigma)\{p.authorizedRedeemer = r\}$

	properly	
PIO.3	ERC20 pools are assigned properly	$\{\}i(o, r, p_{20}, p_{4626}, k, \sigma)\{p.pools20 = p_{20}\}$
PIO.3.1		$\{\}i(o, r, p_{20}, p_{4626}, k, \sigma)\{\forall\pi \in p_{20}: p.pools20Map[\pi.pool] = TokenInfo(true, \pi.n)\}$
PIO.4	ERC4626 pools are assigned properly	$\{\}i(o, r, p_{20}, p_{4626}, k, \sigma)\{p.pools4626 = p_{4626}\}$
PIO.4.1		$\{\}i(o, r, p_{20}, p_{4626}, k, \sigma)\{\forall\pi \in p_{4626}: p.pools4626Map[\pi.pool] = TokenInfo(true, \pi.n)\}$
PIO.5	Pool keeper is assigned properly	$\{\}i(o, r, p_{20}, p_{4626}, k, \sigma)\{p.poolKeeper = k\}$
PIO.6	Supply manager is assigned properly	$\{\}i(o, r, p_{20}, p_{4626}, k, \sigma)\{p.supplyManager = \sigma\}$

Scenario Pool Operations

N	Human description	Formal description
		$\forall p \in MoleculaPool, i \in \aleph, \pi \in Pool, n \in \aleph, k \in A$
		$a_{20} = p.addPool20, s_{20} = p.setPool20, r_{20} = p.removePool20$
		$a_{4626} = p.addPool4626, s_{4626} = p.setPool4626, r_{4626} = p.removePool4626$
		$\kappa = p.setPoolKeeper$

PPI.1	If the scenario addPool20 is not called by the Owner, it fails	$\{msg.sender \neq p.\_owner\}a_{20}(\pi, n)\{S = false\}$
PPI.2	If the pool duplicated, the scenario addPool20 must fail	$\{p.pools20Map[\pi].exist\}a_{20}(\pi, n)\{S = false\}$
PPI.3	Otherwise, the scenario addPool20 is successful	$\{msg.sender = p.\_owner \wedge \neg p.pools20Map[\pi].exist\}a_{20}(\pi, n)\{S = true\}$
PPI.4	If the scenario setPool20 is not called by the Owner, it fails	$\{msg.sender \neq p.\_owner\}s_{20}(i, \pi, n)\{S = false\}$
PPI.5	If the pool duplicated, the scenario setPool20 must fail	$\{p.pools20Map[\pi].exist\}s_{20}(i, \pi, n)\{S = false\}$
PPI.6	If the original pool does not exist, the scenario must fail	$\{p.pools20.length \leq i\}s_{20}(i, \pi, n)\{S = false\}$
PPI.6.1		$\{\pi.pool.balanceOf(p.poolKeeper) \neq 0\}s_{20}(i, \pi, n)\{S = false\}$
PPI.7	Otherwise, the scenario is successful	$\{msg.sender = p.\_owner \wedge \neg p.pools20Map[\pi].exist \wedge p.pools20.length > i\}s_{20}(i, \pi, n)$
		$\{S = true\}$
PPI.8	If the scenario removePool20 is not called by the Owner, it fails	$\{msg.sender \neq p.\_owner\}r_{20}(i)\{S = false\}$
PPI.9	If the original pool does not exist, the scenario must fail	$\{p.pools20.length \leq i\}r_{20}(i)\{S = false\}$

PPI.9.1		$\{p.pools20[i].pool.balanceOf(p.poolKeeper) \neq 0\}r_{20}(i)\{S = false\}$
PPI.10	Otherwise, the scenario is successful	$\{msg.sender = p.\_owner \land p.pools20.length > i \land$
		$p.pools20[i].pool.balanceOf(p.poolKeeper) = 0\}r_{20}(i)\{S = true\}$
PPI.11	If the scenario addPool4626 is not called by the Owner, it fails	$\{msg.sender \neq p.\_owner\}a_{4626}(\pi, n)\{S = false\}$
PPI.12	If the pool duplicated, the scenario addPool4626 must fail	$\{\pi \in p.pools4626Map[\pi].exist\}a_{4626}(\pi, n)\{S = false\}$
PPI.13	Otherwise, the scenario addPool4626 is successful	$\{msg.sender = p.\_owner \land \neg p.pools4626Map[\pi].exist\}a_{4626}(\pi, n)\{S = true\}$
PPI.14	If the scenario setPool4626 is not called by the Owner, it fails	$\{msg.sender \neq p.\_owner\}s_{4626}(i, \pi, n)\{S = false\}$
PPI.15	If the pool duplicated, the scenario setPool4626 must fail	$\{p.pools4626Map[\pi].exist\}s_{4626}(i, \pi, n)\{S = false\}$
PPI.16	If the original pool does not exist, the scenario must fail	$\{p.pools4626.length \leq i\}s_{4626}(i, \pi, n)\{S = false\}$
PPI.16.1		$\{\pi.pool.balanceOf(p.poolKeeper) \neq 0\}s_{4626}(i, \pi, n)\{S = false\}$
PPI.17	Otherwise, the scenario is successful	$\{msg.sender = p.\_owner \land \neg p.pools4626Map[\pi].exist \land p.pools4626.length > i\}s_{4626}(i, \pi, n)$
		$\{S = true\}$

PPI.18	If the scenario removePool4626 is not called by the Owner, it fails	$\{msg.sender \neq p.\_owner\} r_{4626}(i) \{S = false\}$
PPI.19	If the original pool does not exist, the scenario must fail	$\{p.pools4626.length \leq i\} r_{4626}(i) \{S = false\}$
PPI.19.1		$\{p.pools4626[i].pool.balanceOf(p.poolKeeper) \neq 0\} r_{4626}(i) \{S = false\}$
PPI.20	Otherwise, the scenario is successful	$\{msg.sender = p.\_owner \wedge p.pools4626.length > i \wedge$
		$p.pools4626[i].pool.balanceOf(p.poolKeeper) = 0\} r_{4626}(i) \{S = true\}$
PPI.21	Pool Keeper assignment is successful when and only when the sender is the Owner	$\{\} \kappa(k) \{S = (msg.sender = p.\_owner \wedge k \neq 0)\}$
	In case of success:	
PPO.1	ERC20 pool is added	$\dfrac{\{\} a_{20}(\pi,n) \{S=true\}}{\{\forall x: x=p.pools20\} a_{20}(\pi,n) \{p.pools20=x+TokenParams(\pi,n)\}}$
PPO.1.1		$\dfrac{\{\} a_{20}(\pi,n) \{S=true\}}{\{\} a_{20}(\pi,n) \{p.pools20Map[\pi]=TokenInfo(true,n))\}}$
PPO.2	ERC20 pool is substituted	$\dfrac{\{\} s_{20}(i,\pi,n) \{S=true\}}{\{\} s_{20}(i,\pi,n) \{p.pools20[i]=TokenParams(\pi,n)\}}$

PPO.2.1		$\frac{\{\}s_{20}(i,\pi,n)\{S=true\}}{\{\forall\varpi:\varpi=p.pools20[i].pool\wedge\pi\neq\varpi\}s_{20}(i,\pi,n)\{p.pools20Map[\varpi]=TokenInfo(false,0)\}}$
PPO.2.2		$\frac{\{\}s_{20}(i,\pi,n)\{S=true\}}{\{\}s_{20}(i,\pi,n)\{p.pools20Map[\pi]=TokenInfo(true,n)\}}$
PPO.2.3		$\frac{\{\}s_{20}(i,\pi,n)\{S=true\}}{\{\forall\varpi:\varpi=p.pools20[i].pool\wedge\pi=\varpi\wedge x=p.pools20Map\wedge y=p.pools20\}s_{20}(i,\pi,n)\{p.pools20Mapx\wedge p.pools20=y\}}$
PPO.3	ERC20 pool is removed	$\frac{\{\}r_{20}(i)\{S=true\}}{\{\}r_{20}(i)\{\uparrow p.pools20.pop()\}}$
PPO.3.1		$\frac{\{\}r_{20}(i)\{S=true\}}{\{\forall\varpi:\varpi=p.pools20[i].pool\}r_{20}(i)\{\{p.pools20Map[\varpi]=TokenInfo(false,0)\}}$
PPO.3.3		$\frac{\{\}r_{20}(i)\{S=true\}}{\{\forall\varpi:\varpi=p.pools20[p.pools20.length-1].pool\}r_{20}(i)\{\{p.pools20[i]=\varpi\}}$
PPO.4	ERC4626 pool is added	$\frac{\{\}a_{4626}(\pi,n)\{S=true\}}{\{\forall x:x=p.pools4626\}a_{4626}(\pi,n)\{p.pools4626=x+TokenInfo(\pi,n)\}}$
PPO.4.1		$\frac{\{\}a_{4626}(\pi,n)\{S=true\}}{\{\}a_{4626}(\pi,n)\{p.pools4626Map[\pi]=TokenInfo(true,n))\}}$
PPO.5	ERC4626 pool is substituted	$\frac{\{\}s_{4626}(i,\pi,n)\{S=true\}}{\{\}s_{4626}(i\pi,n)\{p.pools4626[i]=TokenParams(\pi,n)\}}$
PPO.5.1		$\frac{\{\}s_{4626}(i,\pi,n)\{S=true\}}{\{\forall\varpi:\varpi=p.pools4626[i].pool\wedge\pi\neq\varpi\}s_{4626}(i,\pi,n)\{p.pools4626Map[\varpi]=TokenInfo(false,0)\}}$
PPO.5.2		$\frac{\{\}s_{20}(i,\pi,n)\{S=true\}}{\{\}s_{20}(i,\pi,n)\{p.pools20Map[\pi]=TokenInfo(true,n)\}}$

PPO.5.3		$\frac{\{\}s_{4626}(i,\pi,n)\{S=true\}}{\{\forall\varpi:\varpi=p.pools4626[i].pool\wedge\pi=\varpi\wedge x=p.pools4626Map\wedge y=p.pools4626\}s_{4626}(i,\pi,n)\{p.pools4626Mapx\wedge p.pools4626=y\}}$
PPO.6	ERC4626 pool is removed	$\frac{\{\}r_{4626}()\{S=true\}}{\{\}r_{4626}()\{\uparrow p.pools4626.pop()\}}$
PPO.6.1		$\frac{\{\}r_{4626}(i)\{S=true\}}{\{\forall\varpi:\varpi=p.pools4626[i].pool\}r_{4626}(i)\{\{p.pools4626Map[\varpi]=TokenInfo(false,0)\}}$
PPO.6.3		$\frac{\{\}r_{4626}(i)\{S=true\}}{\{\forall\varpi:\varpi=p.pools4626[p.pools4626.length-1].pool\}r_{4626}(i)\{\{p.pools4626[i]=\varpi\}}$
PPO.7	Pool Keeper is properly assigned	$\frac{\{\}\kappa(k)\{S=true\}}{\{\}\kappa(k)\{p.poolKeeper=k\}}$

Scenario Set Redeemer

N	Human description	Formal description
		$\forall p \in MoleculaPool, \rho \in A, p = p.setAuthorizedRedeemer$
PRI.1	The scenario is successful if and only if the sender is the owner	$\{\}p(\rho)\{S = (msg.sender = p.\_owner \wedge \rho \neq 0)\}$
	In case of success:	
PRO.1	New Redeemer is assigned	$\frac{\{\}p(\rho)\{S=true\}}{\{\}p(\rho)\{p.authorizedRedeemer=\rho\}}$

Deposit

Scenario SM Deposit

N	Human description	Formal description
		$\forall \sigma \in SupplyManager, t \in A, r \in \aleph, v \in \aleph,\ d = \sigma.deposit$
		$\widehat{d} = \sigma.MOLECULA\_POOL.deposit$
		$\sigma.totalSupply() > 0$
		$\phi = \widehat{d}(t, r, msg.sender, v), \xi = \frac{\phi\sigma.totalSharesSupply}{\sigma.totalSupply()}$
DSI.1	If the sender is not an agent, the scenario fails	$\{\neg\sigma.agents[msg.sender]\}d(t, r, v)\{S = false\}$
DSI.2	Otherwise, the scenario is successful if and only if the Molecula Pool Deposit is successful	$\frac{\{\forall b \in B\}\widehat{d}(t,r,msg.sender,v)\{S=b\}}{\{\sigma.agents[msg.sender]\}d(t,r,v)\{S=b\}}$
	In case of success:	
DSO.1	The scenario returns new shares	$\frac{\{\}d(t,r,v)\{S=true\}}{d(t,r,v)=\xi}$
DSO.2	The total number of shares is increased by the number of new shares issued	$\frac{\{\}d(t,r,v)\{S=true\}}{\{\forall x: x=\sigma.totalSharesSupply\}d(t,r,v)\{\sigma.totalSharesSupply=x+\xi\}}$

DSO.3	The total deposited amount is increased by the new deposit value	$\frac{\{\}d(t,r,v)\{S=true\}}{\{\forall x : x=\sigma.totalDepositedSupply\}d(t,r,v)\{\sigma.totalDepositedSupply=x+\phi\}}$

Scenario MP Deposit

N	Human description	Formal description
		$\forall p \in MoleculaPool, t \in A, r \in \aleph, f \in A, v \in \aleph,\ d = p.deposit$
		$n \in \aleph, z \in \aleph$
		$pool20Map[t].exists \Rightarrow n = pool20Map[t].n \wedge z = v$
		$\neg pool20Map[t].exists \wedge pool4626Map[t].exist \Rightarrow n = pool4626Map[t].n \wedge$
		$z = t.convertToAssets(v)$
		$\phi = 10^{n} z$
DPI.1	If the sender is not a Supply Manager, the scenario must fail	$\{msg.sender \neq p.supplyManager\}d(t, r, f, v)\{S = false\}$
DPI.2	If the pool is unknown, the scenario must fail	$\{\neg pools20Map[t].exist \wedge \neg pools4626Map[t].exist\}d(t, r, f, v)\{S = false\}$
DPI.3	If transfer fails, the scenario must fail	$\{\}t.safeTransferFrom(f, p.poolKeeper, v)\{S = false\} \Rightarrow \{\}d(t, r, f, v)\{S = false\}$

DPI.4	Otherwise, the scenario is successful	$\frac{\{\}t.safeTransferFrom(f,p.poolKeeper,v)\{S=true\}}{\{msg.sender=p.supplyManager\wedge(pools20Map[t].exist\vee t\in p.pools4626.pool)\}d(t,r,f,v)\{S=true\}}$
	In case of success:	
DPO.1	Transfer message is sent	$\frac{\{\}d(t,r,f,v)\{S=true\}}{\{\}d(t,r,f,v)\{\uparrow t.safeTransferFrom(f,p.poolKeeper,v)\}}$
DPO.2	Scenario returns a normalized (to 18-decimals) value	$\frac{\{\}d(t,r,f,v)\{S=true\}}{d(t,r,f,v)=\phi}$

Scenario Token Deposit Request

N	Human description	Formal description
		$\forall\tau \in RebaseTokenCommon,\ a \in \aleph256, c \in A,\ o \in A, r = 'requestDeposit'$
		$i = \tau.\_generateOperationId()$
		$\kappa = (s = o \vee \tau.isOperator[o][s]) \wedge a \geq \tau.minDepositValue \wedge c \neq 0 \wedge o \neq 0 \wedge$
		$(\tau.depositRequests[i].status = OperationStatus.None)$
DTI.1	In case the sender neither the owner nor the operator, scenario fails	$\{msg.sender \neq o \wedge \neg\tau.isOperator[o][msg.sender]\}\tau.r(a, c, o)\{S = false\}$
DTI.2	In case of too low deposit, the scenario fails	$\{a < \tau.minDepositValue\}\tau.r(a, c, o)\{S = false\}$
DTI.3	In case of zero controller or owner, the scenario must fail	$\{c = 0 \vee o = 0\}\tau.r(a, c, o)\{S = false\}$

DTI.4	In case of known deposit, the scenario must fail	$\{\tau.depositRequests[i].status \neq OperationStatus.None\}\tau.r(a,c,o)\{S = false\}$
DTI.5	Otherwise, the scenario is successful if and only if, the underlying scenario is successful	$\frac{\{\}\tau.accountant.requestDeposit(i,o,a)\{S=true\}}{\{\kappa\}\tau.r(a,c,o)\{S=true\}}$
	In case of success:	
DTO.1		$\frac{\{\}\tau.r(a,c,o)\{S=true\}}{\{\}\tau.r(a,c,o)\{\uparrow\tau.accountant.requestDeposit(i,o,a)\}}$
DTO.1.1		$\frac{\{\}\tau.r(a,c,o)\{S=true\}}{\{\}\tau.r(a,c,o)\{\tau.depositRequests[i].status=OperationStatus.Pending\vert}$

Scenario Token Deposit Confirm

N	Human description	Formal description
		$\forall\tau \in RebaseTokenCommon,\ i \in \aleph256, \sigma \in \aleph256, c = 'confirmDeposit'$
DCI.1	In case the scenario is not initiated by 'Accountant Deposit' it fails	$\{msg.sender \neq \tau.accountant\}c(i,\sigma)\{S = false\}$
DCI.2	In case of zero recipient the scenario must fail	$\{\tau.depositRequests[i].addr = 0\}c(i,\sigma)\{S = false\}$
DCI.3	In case of unknown request the scenario must fail	$\{\tau.depositRequests[i].status \neq OperationStatus.Pending\}c(i,\sigma)\{S = false\}$

DCI.4	Otherwise, the scenario is successful	$\{msg.sender = \tau.accountant \wedge \tau.depositRequests[i].addr \neq 0 \wedge$
		$\tau.depositRequests[i].status = OperationStatus.Pending\}c(i, \sigma)\{S = true\}$
	In case of success:	
DCO.1	Number of shares owned by the recipient is increased by the request amount	$\frac{\{\}c(i,\sigma)\{S=true\}}{\{\forall x{:}x=\tau.shares[\tau.depositRequests[i].addr]\}c(i,\sigma)\{\tau.shares[\tau.depositRequests[i].addr]=x+\sigma\}}$
DCO.2	Deposit request is marked as confirmed	$\frac{\{\}c(i,\sigma)\{S=true\}}{\{\}c(i,\sigma)\{\tau.depositRequests[i].status=OperationStatus.Confirmed\}}$

Redeem

Scenario SM Redeem Request

N	Human description	Formal description
		$\forall\sigma \in SupplyManager, t \in A, r \in \aleph, \varsigma \in \aleph, \rho = \sigma.requestRedeem$
		$\hat{\rho} = \sigma.MOLECULA\_POOL.requestRedeem$
		$\hat{v} = \frac{\varsigma\sigma.totalSupply()}{\sigma.totalSharesSupply}$
		$\bar{y}_0 \in \aleph$
		$\Delta = (\sigma.apyFormatter \neq 0) \wedge (\sigma.totalDepositedSupply < \sigma.totalSupply())$

		$\Delta \Rightarrow \overline{y}_0 = \frac{\varsigma(\sigma.totalSupply() - \sigma.totalDepositedSupply)(APY\_FACTOR - \sigma.apyFormatter)}{\sigma.totalSharesSupply \cdot \sigma.apyFormatter}$
		$\neg\Delta \Rightarrow \overline{y}_0 = 0$
		$\overline{y}_s = \frac{\overline{y}_0 \sigma.totalSharesSupply}{\sigma.totalSupply()}$
		$\vartheta = \widehat{\rho}(t, \widehat{v})$
		$\sigma.totalSharesSupply > 0$
RSI.1	If the scenario is called by not an agent, it fails	$\{msg.sender \notin \sigma.agents\}\rho(t, r, \varsigma)\{S = false\}$
RSI.2	If the request is too big, the scenario must fail	$\{\varsigma > \sigma.totalSharesSupply\}\rho(t, r, \varsigma)\{S = false\}$
RSI.3	If the request already exists, the scenario must fail	$\{\sigma.redeemRequests[r].status \neq OperationStatus.None\}\rho(t, r, \varsigma)\{S = false\}$
RSI.4	If the underlying Molecula Pool Request Redeem scenario fails, the scenario fails	$\{\}\ \widehat{\rho}(t, \widehat{v})\{S = false\} \Rightarrow \{\}\rho(t, r, \varsigma)\{S = false\}$
RSI.5	Otherwise, the scenario is successful	$\frac{\{\}\widehat{\rho}(t,\widehat{v})\{S=true\}}{\{msg.sender \in \sigma.agents \wedge \varsigma \le \sigma.totalSharesSupply \wedge \sigma.redeemRequests[r].status = OperationStatus.None\}\rho(t,r,\varsigma)\{S=true\}}$
	In case of success:	
RSO.1	Locked value is increased by the requested amount of	$\frac{\{\}\rho(t,r,\varsigma)\{S=true\}}{\{\forall x: x=\sigma.lockedYieldShares\}\rho(t,r,\varsigma)\{\sigma.lockedYieldShares=x+\overline{y}_s\}}$

	shares	
RSO.2	Total deposit amount is changed accordingly	$\dfrac{\{\}\rho(t,r,\varsigma)\{S=true\}}{\{\forall x,z:x=\sigma.totalDepositedSupply\wedge z=\sigma.totalSharesSupply\}\rho(t,r,\varsigma)\{\sigma.totalDepositedSupply=x+\overline{y}_0-\frac{\varsigma x}{z}\}}$
RSO.3	Total shares are changes accordingly	$\dfrac{\{\}\rho(t,r,\varsigma)\{S=true\}}{\{\forall z:z=\sigma.totalSharesSupply\}\rho(t,r,\varsigma)\{\sigma.totalSharesSupply=z+\overline{y}_s-\varsigma\}}$
RSO.4	Molecula Pool Request Redeem subscenario is invoked	$\dfrac{\{\}\rho(t,r,\varsigma)\{S=true\}}{\{\}\rho(t,r,\varsigma)\{\uparrow\widehat{p}(t,\widehat{v})\}}$
RSO.5	New request is registered and assigned by the value returned from Molecula Pool	$\dfrac{\{\}\rho(t,r,\varsigma)\{S=true\}}{\{\}\rho(t,r,\varsigma)\{\sigma.redeemRequests[r]=RedeemOperationInfo(msg.sender,\vartheta,OperationStatus.Pending)\}}$
RSO.6	The scenario returns provided from the underlying Molecula Pool Redeem Request Scenario	$\dfrac{\{\}\rho(t,r,\varsigma)\{S=true\}}{\rho(t,r,\varsigma)=\vartheta}$

Scenario MP Redeem Request

N	Human description	Formal description
		$\forall p\in MoleculaPool, t\in A, v\in\aleph,\ \rho=p.requestRedeem, k\in\aleph$
		$p.poolsMap20[t].exist\Rightarrow k=v10^{-p.pools20Map[t].n}$
		$\neg p.poolsMap20[t].exist\wedge p.poolsMap4626[t].exist\Rightarrow$

		$k = t.convertToShares\left(v10^{-p.pools4626Map[t].n}\right)$
RPI.1	If the scenario is invoked not by SM Request Redeem superscenario, it fails	$\{msg.sender \neq p.supplyManager\}\rho(t, v)\{S = false\}$
RPI.2	If the pool is unknown the scenario fails	$\{\neg p.poolsMap20[t].exist \land \neg p.poolsMap4626[t].exist\}\rho(t, v)\{S = false\}$
RPI.3	Otherwise, the scenario is successful	$\{msg.sender = p.supplyManager \land (p.poolsMap20[t].exist \lor p.poolsMap4626[t].exist)\}$
		$\rho(t, v)\{S = false\}$
	In case of success:	
RPO.1	Value to redeem is increased by value	$\frac{\{\}\rho(t,v)\{S=true\}}{\{\forall x: x=p.valueToRedeem\}\rho(t,v)\{p.valueToRedeem=x+v\}}$
RPO.2	Denormalized value is returned	$\frac{\{\}\rho(t,v)\{S=true\}}{\rho(t,v)=k}$

Scenario Token Request Redeem

N	Human description	Formal description
		$\forall\tau \in RebaseTokenCommon,\ \sigma \in \aleph256,\ c \in A,\ o \in A, r = 'requestRedeem'$
		$i = \tau.\_generateOperationId()$
		$\hat{\sigma} = min(\sigma, \tau.sharesOf(o))$

		$\kappa = (msg.sender = o \vee \tau.isOperator[o][msg.sender]) \wedge$
		$(\hat{\sigma} \geq \tau.minRedeemValue) \wedge$
		$(\tau.redeemRequests[i].status = OperationStatus.None) \wedge$
		$(\tau.sharesOf(o) > 0) \wedge (o \neq 0) \wedge (c \neq 0)$
RTI.1	If the scenario is called neither by the owner, or the operator, it fails	$\{msg.sender \neq o \wedge \neg\tau.isOperator[o][msg.sender]\}\tau.r(\sigma, c, o)\{S = false\}$
RTI.2	If the request is known, the scenario must fail	$\{\tau.redeemRequests[i].status \neq OperationStatus.None\}\tau.r(\sigma, c, o)\{S = false\}$
RTI.3	If the request is too small, the scenario must fail	$\{\hat{\sigma} < \tau.minRedeemValue\}\tau.r(\sigma, c, o)\{S = false\}$
RTI.4	If the balance is zero, the scenario must fail	$\{\tau.sharesOf(o) = 0\}\tau.r(\sigma, c, o)\{S = false\}$
RTI.5	If either owner or controller is zero, the scenario must fail	$\{o = 0 \vee c = 0\}\tau.r(\sigma, c, o)\{S = false\}$
RTI.6	Otherwise, the scenario is successful if any only if, the underlying scenario is successful	$\dfrac{\{\}\tau.accountant.requestRedeem\{value{:}msg.value\}(i,\sigma)\{S=true\}}{\{\kappa\}\tau.r(\sigma,c,o)\{S=true\}}$
	In case of success:	
RTO.1	The proper amount of shares is burnt from owner's account	$\dfrac{\{\}\tau.r(\sigma,c,o)\{S=true\}}{\{\forall x{:}x=\tau.sharesOf(o)\}\tau.r(\sigma,c,o)\{\tau.sharesOf(o)=x-\hat{\sigma}\}}$

RTO.2	The request becomes in progress	$\frac{\{\}\tau.r(\sigma,c,o)\{S=true\}}{\{\}\tau.r(\sigma,c,o)\{\tau.redeemRequests[i].status=OperationStatus.Pending\}}$
RTO.3	The scenario is a composition of 'prepare' and the underlying scenario	$\frac{\{\}\tau.r(\sigma,c,o)\{S=true\}}{\{\}\tau.r(\sigma,c,o)\{\uparrow\tau.accountant.requestRedeem\{value:msg.value\}(i,\widehat{\sigma})\}}$

Scenario MP Redeem

N	Human description	Formal description
		$\forall p \in MoleculaPool, r \subset \aleph 256,\ w = p.redeem, k \in \aleph$
		$\widehat{w} = p.supplyManager.redeem\{value = msg.value\}$
		$p.poolsMap20[t].exist \Rightarrow k = v10^{p.pools20Map[t].n}$
		$\neg p.poolsMap20[t].exist \wedge p.poolsMap4626[t].exist \Rightarrow k = v10^{p.pools4626Map[t].n}$
WPI.1	If the invoker is not the redeemer, the scenario fails	$\{msg.sender \neq p.authorizedRedeemer\}w(r)\{S = false\}$
WPI.2	If the request is empty, the scenario fails	$\{r = \varnothing\}w(r)\{S = false\}$
WPI.3	If the SMRedeem scenario fails, this scenario fails as well	$\{\}\ \widehat{w}(p.poolKeeper, r)\{S = false\} \Rightarrow \{\}w(r)\{S = false\}$
WPI.4	Otherwise, the scenario is successful	$\frac{\{\}\widehat{w}(p.poolKeeper,r)\{S=true\}}{\{msg.sender=p.authorizedRedeemer \wedge r \neq \varnothing\}w(r)\{S=true\}}$

	In case of success:	
WPO.1	In case of success, the value to redeem is updated by the normalized value	$\frac{\{\}w(r)\{S=true\}}{\{\forall x{:}x=p.valueToRedeem\}w(r)\{p.valueToRedeem=x-k\}}$

SM Redeem

N	Human description	Formal description
		$\forall\sigma \in SupplyManager, k \in A, r \subset \aleph256,\ w = \sigma.redeem, \rho = \sigma.redeemRequests$
		$\Xi = (msg.sender = \sigma.MOLECULA\_POOL) \land$
		$(\forall i{:}\, i \in r \land \rho[i].status = OperationStatus.Pending) \land$
		$(\forall i, j{:}\, i \in r \land j \in r \land \rho[i].agent = \rho[j].agent) \land$
		$\forall i{:}\, i \in r \land \rho[i].agent \in \sigma.agents$
		$a \in \aleph,\ \exists i{:}\, i \in r \land a = \rho[i].agent$
		$v = \sum_{i \in r} \rho[i].value$
		$\forall x \in r{:} \{\}x.getERC20Token()\{S = true\}$
WSI.1	If the the scenario invoked not by MP Redeem	$\{msg.sender \neq \sigma.MOLECULA\_POOL\}w(k, r)\{S = false\}$

	superscenario, it fails	
WSI.2	If there is an unknown request, the scenario fails	$\{\exists i: i \in r \wedge \rho[i].status \neq OperationStatus.Pending\}w(k,r)\{S = false\}$
WSI.3	If different agents are used, the scenario must fail	$\{\exists i,j: i \in r \wedge j \in r \wedge \rho[i].agent \neq \rho[j].agent\}w(k,r)\{S = false\}$
WSI.4	If there is a request with an unknown agent, the scenario must fail	$\{\exists i: i \in r \wedge \rho[i].agent \notin \sigma.agents\}w(k,r)\{S = false\}$
WSI.5	Otherwise, the scenario is successful if and only if agent's redeem is successful	$\frac{\{\}a.redeem\{value{:}msg.value\}(k,r,redeemRequests[r].value,v)\{S=true\}}{\{\Xi\}w(k,r)\{S=true\}}$
	In case of success:	
WSO.1	All the requests are removed from the list	$\frac{\{\}w(k,r)\{S=true\}}{\{\}w(k,r)\{\forall i\in r{:}r[i].status=OperationStatus.Confirmed\}}$
WSO.2	Total value is a sum of all the request values	$\frac{\{\}w(k,r)\{S=true\}}{w(k,r)=\{a.getERC20Token(),v\}}$
WSO.3	Redeem message is sent to the agent with proper values	$\frac{\{\}w(k,r)\{S=true\}}{\{\}w(k,r)\{\uparrow a.redeem\{value{:}msg.value\}(k,r,redeemRequests[r].value,v)\}}$

Scenario Token Redeem

N	Human description	Formal description

		$\forall \tau \in RebaseTokenCommon,\ \rho \in [\aleph 256],\ v \in [\aleph 256],\ r = 'redeem'$
		$i \in \aleph 256,\ i < \rho.length,\ \rho.length = v.length,\ \hat{\rho} = \tau.redeemRequests[\rho[i]]$
		$\hat{\rho}.status = OperationStatus.Pending$
		$J \subset [0..\rho.length - 1],\ \forall k \in [0..\rho.length - 1]: k \in J \Leftrightarrow \tau.redeemRequests[\rho[k]].status =$
		$OperationStatus.Pending$
WTI.1	The scenario is successful if and only if it is initiated by 'Accountant Redeem' superscenario	$\{\}\tau.r(\rho, v)\{S = (msg.sender = \tau.accountant)\}$
	In case of success:	
WTO.1	All the participating requests are removed from redeem requests	$\frac{\{\}\tau.r(\rho,v)\{S=true\}}{\{\}\tau.r(\rho,v)\{\hat{\rho}.status=OperationStatus.ReadyToConfirm\}}$
WTO.2	All the partcipating requests are added to 'ready to confirm list'	Covered by WTO.1
WTO.3	The value of each participating request is set according to the values provided	$\frac{\{\}\tau.r(\rho,v)\{S=true\}}{\{\}\tau.r(\rho,v)\{\hat{\rho}.value=v[i]\}}$
WTO.4	The total value of all the participating requests is returned	$\frac{\{\}\tau.r(\rho,v)\{S=true\}}{\tau.r(\rho,v)=\sum_{j\in J} v[j]}$

Scenario Token Redeem Confirm

N	Human description	Formal description
		$\forall\tau \in RebaseTokenCommon,\ i \in \aleph256,\ r = 'confirmRedeem'$
		$\rho = \tau.redeemRequests[i]$
OTI.1	The scenario is successful if and only if the request is known	$\frac{\{\}\tau.accountant.confirmRedeem(\rho.addr,\rho.val)\{S=true\}}{\{\rho.status=OperationStatus.ReadyToConfirm\}\tau.r(i)\{S=true\}}$
OTI.1.1		$\{\rho.status \neq OperationStatus.ReadyToConfirm\}\tau.r(i)\{S = false\}$
	In case of success:	
OTO.1	Update removes the request from 'ready to confirm' list	$\frac{\{\}\tau.r(i)\{S=true\}}{\{\}\tau.r(i)\{\rho.status=OperationStatusConfirmed\}}$
OTO.2	The scenario is a superposition of 'update' and the underlying scenario	$\frac{\{\}\tau.r(i)\{S=true\}}{\{\}\tau.r(i)\{\uparrow\tau.accountant.confirmRedeem(\rho.addr,\rho.val)\}}$

Token

Scenario Mint

N	Human description	Formal description
		$\forall\tau \in RebaseERC20, t \in A, a \in \aleph256,\ m = 'mint'$

TMI.1	If the sender is not an accountant, the scenario fails	$\{msg.sender \neq \tau.\_owner\}\tau.m(t, a)\{S = false\}$
TMI.2	If the recipient is null, the scenario must fail	$\{t = 0\}\tau.m(t, a)\{S = false\}$
TMI.3	Otherwise, the scenario is successful	$\{msg.sender = \tau.owner \wedge t \neq 0\}\tau.m(t, a)\{S = true\}$
	In case of success:	
TMO.1	A number of shares for the recipient is increased accordingly	$\frac{\{\}\tau.m(t,a)\{S=true\}}{\{\forall x:x=\tau.\_shares[t]\}\tau.m(t,a)\{\tau.\_shares[t]=x+a\}}$

Scenario Transfer

N	Human description	Formal description
		$\forall\tau \in RebaseERC20, t \in A, a \in \aleph256,\ \hat{t} = 'transfer'$
		$\alpha = convertToShares(a)$
TTI.1	If the sender is zero, the scenario must fail	Never occurs
TTI.2	If the recipient is zero, the scenario must fail	$\{t = 0\}\tau.\hat{t}(t, a)\{S = false\}$
TTI.3	If the sender does not have enough assets, the scenario	$\{\tau.balanceOf(msg.sender) < a\}\tau.\hat{t}(t, a)\{S = false\}$

	must fail	
TTI.4	Otherwise, the scenario is successful	$\{\tau.balanceOf(msg.sender) \geq a \wedge t \neq 0\}\tau.\hat{t}(t, a)\{S = true\}$
	In case of success:	
TTO.1	Sender's balance is decreased accordingly	$\frac{\{\}\tau.\hat{t}(t,a)\{S=true\}}{\{\forall x: x=\tau.balanceOf(msg.sender) \wedge msg.sender \neq t\}\tau.\hat{t}(t,a)\{\tau.balanceOf(msg.sender)=x-a\}}$
TTO.2	Recipient's balance is increased accordingly	$\frac{\{\}\tau.\hat{t}(t,a)\{S=true\}}{\{\forall x: x=\tau.balanceOf(t) \wedge msg.sender \neq t\}\tau.\hat{t}(t,a)\{\tau.balanceOf(t)=x+a\}}$

Scenario TransferFrom

N	Human description	Formal description
		$\forall \tau \in RebaseERC20, f \in A, t \in A, a \in \aleph 256,\ \hat{t} = 'transferFrom'$
		$\alpha = convertToShares(a)$
TFI.1	If the sender is zero, the scenario must fail	$\{f = 0\}\tau.\hat{t}(f, t, a)\{S = false\}$
TFI.2	If the recipient is zero, the scenario must fail	$\{t = 0\}\tau.\hat{t}(f, t, a)\{S = false\}$
TFI.3	If the sender does not have enough assets, the scenario must fail	$\{\tau.balanceOf(f) < a\}\tau.\hat{t}(f, t, a)\{S = false\}$

TFI.4	If the sender does not have enough allowance, the scenario must fail	$\{\tau.allowance(f, msg.sender) < a\}\tau.\hat{t}(f, t, a)\{S = false\}$
TFI.5	Otherwise, the scenario is successful	$\{\tau.balanceOf(f) \geq a \wedge t \neq 0 \wedge f \neq 0 \wedge \tau.allowance(f, msg.sender) \geq a \wedge$
		$\tau.balanceOf(f) \geq a\}\tau.\hat{t}(f, t, a)\{S = true\}$
	In case of success:	
TFO.1	Sender's balance is decreased accordingly	$\frac{\{\}\tau.\hat{t}(f,t,a)\{S=true\}}{\{\forall x:x=\tau.balanceOf(f)\}\tau.\hat{t}(f,t,a)\{\tau.balanceOf(f)=x-a\}}$
TFO.2	Recipient's balance is increased accordingly	$\frac{\{\}\tau.\hat{t}(f,t,a)\{S=true\}}{\{\forall x:x=\tau.balanceOf(t)\}\tau.\hat{t}(f,t,a)\{\tau.balanceOf(t)=x+a\}}$
TFO.3	Allowance is decreased accordingly	$\frac{\{\}\tau.\hat{t}(f,t,a)\{S=true\}}{\{\forall x:x=\tau.allowance(f,msg.sender)\wedge x<2^{256}-1\}\tau.\hat{t}(f,t,a)\{\tau.allowance(f,msg.sender)=x-a\}}$

Scenario Approve

N	Human description	Formal description
		$\forall\tau \in RebaseERC20, t \in A, a \in \aleph256,\ \hat{a} = 'approve'$
TAI.1	In case of zero allower, the scenario fails	Deprecated
TAI.2	In case of zero spender, the scenario fails	$\{t = 0\}\tau.\hat{a}(t, a)\{S = false\}$

TAI.3	Otherwise, the scenario is successful	$\{t \neq 0\}\tau.\hat{a}(t, a)\{S = true\}$
TAI.3.1		$\{t \neq 0\}\tau.\hat{a}(t, 2^{256} - 1)\{S = true\}$
	In case of success:	
TAO.1	Allowance is set	$\frac{\{\}\tau.\hat{a}(t,a)\{S=true\}}{\{\}\tau.\hat{a}(t,a)\{\tau.allowance(msg.sender,t)=a\}}$

Scenario Setters

N	Human description	Formal description
		$\forall\tau \in RebaseTokenCommon, r \in A, a \in A,\ m_r \in \aleph256,\ m_s \in \aleph256$
		$\hat{r}$ = $'setOracle', \hat{a} = 'setAccountant',\ \hat{m}_d = 'setMinDepositValue', \hat{m}_r = 'setMinRedeemValue'$
TSI.1	setOracle is successful if and only if it is called by the Owner	$\{\}\ \hat{r}(r)\{S = (msg.sender = \tau.\_owner \wedge r \neq 0)\}$
TSI.2	setAccountant is successful if and only if it is called by the Owner	$\{\}\ \hat{a}(a)\{S = (msg.sender = \tau.\_owner \wedge a \neq 0)\}$
TSI.3	setMinDepositValue is successful if and only if it is called by the Owner	$\{\}\hat{m}_d(m_d)\{S = (msg.sender = \tau.\_owner \wedge m_d > 0)\}$

TSI.4	setMinRedeemValue is successful if and only if it is called by the Owner	$\{\}\hat{m}_r(m_r)\{S = (msg.sender = \tau.\_owner \wedge m_r > 0)\}$
	In case of success:	
TSO.1	setOracle sets oracle	$\frac{\{\}\hat{r}(r)\{S=true\}}{\{\}\hat{r}(r)\{\tau.oracle=r\}}$
TSO.2	setAccountant sets accountant	$\frac{\{\}\hat{a}(a)\{S=true\}}{\{\}\hat{a}(a)\{\tau.accountant=a\}}$
TSO.3	setMinDepositValue sets minimal deposit value	$\frac{\{\}\hat{m}_d(m_d)\{S=true\}}{\{\}\hat{m}_d(m_d)\{\tau.minDepositValue=m_d\}}$
TSO.4	setMinRedeemValue sets minimal redeem value	$\frac{\{\}\hat{m}_r(m_r)\{S=true\}}{\{\}\hat{m}_r(m_r)\{\tau.minRedeemValue=m_r\}}$

Molecula Pool 2 Set Of scenarios

Deposit

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, t \in A, r \in \aleph256, f \in A, v \in \aleph256$
		$d = \text{'}deposit\text{'},\ \sigma = p.SUPPLY\_MANAGER$
		$\kappa = t.safeTransferFrom(f, p, v)$

		$\phi \in \aleph 256$
		$p.poolMap[t].tokenType = ERC20 \Rightarrow \phi = \_normalize(p.poolMap[t].n, v)$
		$p.poolMap[t].tokenType = ERC4626 \Rightarrow$
		$\phi = \_normalize(p.poolMap[t].n, t.convertToAssets(v))$
DPJ.1	If the sender is not a Supply Manager, the scenario must fail	$\{s \neq \sigma\}p.d(t, r, f, v)\{S = false\}$
DPJ.2	If the pool is unknown, the scenario must fail	$\{p.poolMap[t].tokenType = None\}p.d(t, r, f, v)\{S = false\}$
DPJ.3	If transfer fails, the scenario must fail	$\{\}\kappa\{S = false\} \Rightarrow \{\}p.d(t, r, f, v)\{S = false\}$
DPJ.4	Otherwise, the scenario is successful	$\frac{\{\}\kappa\{S=true\}}{\{s=\sigma \wedge (p.poolMap[t].tokenType \neq None)\}p.d(t,r,f,v)\{S=true\}}$
	In case of success:	
DPN.1	Transfer message is sent	$\frac{\{\}p.d(t,r,f,v)\{S=true\}}{\{\}.p.d(t,r,f,v)\{\uparrow\kappa\}}$
DPN.2	Scenario returns a normalized (to 18-decimals) value	$\frac{\{\}p.d(t,r,f,v)\{S=true\}}{p.d(t,r,f,v)=\phi}$

Execute

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, t \in A, d \in [byte],$
		$d.length >= 36$
		$e = \text{'}execute\text{'}, k = p.poolKeeper, \sigma = d[32..36), \varsigma = d[36..56), \alpha = d[56..88)$
		$a = (\sigma = IERC20.approve.selector)$
		$a \Rightarrow d.length \geq 88, \varsigma \neq 0$
		$\kappa = \neg p.isExecutePaused \wedge \neg p.poolMap[t].isBlocked$
EXJ.1	If the scenario is not initiated by Pool Keeper, it fails	$\{s \neq k\}p.e(t, d)\{S = false\}$
EXJ.2	If the scenario is about approve and the value is not zero, the scenario fails	$\{a \wedge \$(msg) \neq 0\}p.e(t, d)\{S = false\}$
EXJ.3	If the scenario is about approve and the target address is not whitelisted, the scenario fails	$\{a \wedge \neg p.isInWhiteList[\varsigma]\}p.e(t, d)\{S = false\}$
EXJ.3a	If the execution is paused, the scenario fails	$\{p.isExecutePaused\}p.e(t, d)\{S = false\}$
EXJ.3b	If the target corresponds to the blocked token, the scenario fails	$\{p.poolMap[t].isBlocked\}p.e(t, d)\{S = false\}$

EXJ.4	Otherwise, the approve scenario is successful	$\{s = \kappa \wedge k \wedge a \wedge \$(msg = 0) \wedge p.isInWhiteList[\varsigma]\}p.e(t, d)\{S = true\}$
EXJ.5	If the operation is not an approve and the target address is not whitelisted, the scenario fails	$\{\neg a \wedge \neg p.isInWhiteList[t]\}p.e(t, d)\{S = false\}$
EXJ.6	if the operation is not an approve, the scenario succeeds	$\{s = \kappa \wedge k \wedge \neg a \wedge p.isInWhiteList[t]\}p.e(t, d)\{S = true\}$
	In case of success:	
EXN.1	In case of approve, the allowance is set accordingly	$\frac{\{\}p.e(t,d)\{S=true\wedge a\}}{\{\}p.e(t,d)\{t.allowance(p,\varsigma)=\alpha\}}$
EXN.2	In case of not approve, the corresponding message is sent	$\frac{\{\}p.e(t,d)\{S=true\wedge\neg a\}}{\{\}p.e(t,d)\{\uparrow t.functionalCallWithValue(d,\$(msg))\}}$

Init

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, o \in A, p \subset A$
		$k \in A, \sigma \in A, w \subset A, u \in A, g \in A$
		$i = 'constructor', \kappa = (\forall \pi_1, \pi_2 \in p: \pi_1 \neq \pi_2 \Rightarrow \pi_1.token \neq \pi_2.token)$

		$\kappa_p = (\forall\pi \in p : \pi \in ERC20 \vee \pi \in ERC4626)$
		$\kappa_1 = \forall(\omega \in w): \omega \neq 0$
PIJ.1	Scenario is always successful if and only if no parameters are zero	$\{\}p.i(o, p, k, \sigma, w, g)\{S = \kappa \wedge \kappa_1 \wedge \kappa_p \wedge (o \neq 0) \wedge (k \neq 0) \wedge (\sigma \neq 0) \wedge (g \neq 0)\}$
	In case of success:	
PIN.1	Owner is assigned properly	$\frac{\{\}p.i(o,p,k,\sigma,w,g)\{S=true\}}{\{\}p.i(o,p,k,\sigma,w,g)\{p.\_owner=o\}}$
PIN.2	Whitelist is assigned properly	$\frac{\{\}p.i(o,p,k,\sigma,w,g)\{S=true\}}{\{\}p.i(o,p,k,\sigma,w,g)\{\forall\omega\in w:p.\ isInWhiteList[\omega] = true\}}$
PIN.3	Pools are assigned properly	$\frac{\{\}p.i(o,p,k,\sigma,w,g)\{S=true\}}{\{\}p.i(o,p,k,\sigma,w,g)\{\forall\pi:\pi\in p:\uparrow p.\_addToken(\pi)\}}$
PIN.5	Pool keeper is assigned properly	$\frac{\{\}p.i(o,p,k,\sigma,w,g)\{S=true\}}{\{\}p.i(o,p,k,\sigma,w,g)\{p.\_poolKeeper=k\}}$
PIN.6	Supply manager is assigned properly	$\frac{\{\}p.i(o,p,k,\sigma,w,g)\{S=true\}}{\{\}p.i(o,p,k,\sigma,w,g)\{p.\_SUPPLY\_MANAGER=\sigma\}}$
PIN.7	Guardian is assigned properly	$\frac{\{\}p.i(o,p,k,\sigma,w,g)\{S=true\}}{\{\}p.i(o,p,k,\sigma,w,g)\{p.guardian=g\}}$

Migrate

N	Human description	Formal description

		$\forall p \in MoleculaPoolTreasury, o \in MoleculaPool$
		$m = \text{'}migrate\text{'}, \sigma = p.SUPPLY\_MANAGER$
		$\kappa = \forall t: t \in (o.getPool20() \cup o.getPool4626()) \Rightarrow t.allowance(o.poolKeeper(), p) \geq$
		$t.balanceOf(o.poolKeeper())$
		$\delta = \forall \pi \in (o.pool20 \cup 0.pool4626): \pi.valueToRedeem < 5 \cdot 10^{17}$
		$\forall \pi_1, \pi_2 \in (o.pool20 \cup o.pool4626): \pi_1 \neq \pi_2 \Rightarrow \pi_1.pool \neq \pi_2.pool$
		$\forall \pi_1 \in p.pool, \pi_2 \in o.pool4626: \pi_1.token \neq \pi_2.pool$
		$\forall \pi_1 \in p.pool, \pi_2 \in o.pool20: \pi_1.token \neq \pi_2.pool$
		$\forall \pi \in o.pool20: \pi \in ERC20 \land \pi \notin ERC4626$
		$\forall \pi \in o.pool4626: \pi \in ERC20 \land \pi \in ERC4626$
PMJ.1	If the scenario is not initiated by Supply Manager, it fails	$\{s \neq \sigma\}p.m(o)\{S = false\}$
PMJ.1a	If any old pool has a high value to redeem, the scenario fails	$\{\neg\delta\}p.m(o)\{S = false\}$
PMJ.1.1	In case of insufficient allowance, the scenario fails	$\{\neg\kappa\}p.m(o)\{S = false\}$
PMJ.2	Otherwise, the scenario is	$\{s = \sigma \land \kappa \land \delta\}p.m(o)\{S = true\}$

	successful	
	In case of success:	
PMN.2	All the ERC20 pools are transferred	$\frac{\{\}p.m(o)\{S=true\}}{\{\forall\pi:\pi\in o.pool20\}p.m(o)\{\pi\in p.pool\}}$
PMN.3	All the ERC4626 pools are transferred	$\frac{\{\}p.m(o)\{S=true\}}{\{\forall\pi:\pi\in o.pool4626\}p.m(o)\{\pi\in p.pool\}}$
PMN.4	All the assets are transferred to the new pools	$\frac{\{\}p.m(o)\{S=true\}}{\{\forall\pi:\pi\in(o.pool4626)\vee\pi\in(o.pool20)\wedge x=\pi.balanceOf(o.poolKeeeper)\wedge y=\pi.balanceOf(p)\}p.m(o)\{\pi.balanceOf(p)=x+y\}}$
PMN.5	Unlimited allowance is set for all the agents	$\frac{\{\}p.m(o)\{S=true\}}{\{\forall a:a\in\sigma.getAgents()\}p.m(o)\{a.getERC20Token().allowance(p,a)=2^{256}-1\}}$

Pool Operations

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, i \in \aleph, \pi \in A, k \in A$
		$n(a) \in A \rightarrow Z = 18 - decimals(a)$
		$a = p.addToken, r = p.removeToken$
		$\kappa = p.setPoolKeeper$
		$\Xi_{20} = \pi \in ERC20 \wedge \pi \notin ERC4626$

		$\Xi_{4626} = \pi \in ERC20 \ \wedge \ \pi \in ERC4626$
		$\Omega = p.poolMap[\pi].tokenType = None$
PPJ.1	If the scenario addToken is not called by the Owner, it fails	$\{msg.sender \neq p.\_owner\}p.a(\pi)\{S = false\}$
PPJ.2	If the pool duplicated, the scenario addToken must fail	$\{\neg\Omega\}p.a(\pi)\{S = false\}$
PPJ.2.1		$\{\neg\Xi_{20} \wedge \neg\Xi_{4626}\}p.a(\pi)\{S = false\}$
PPJ.3	Otherwise, the scenario addToken is successful	$\{msg.sender = p.\_owner \wedge \left(\Xi_{20} \vee \Xi_{4626}\right) \wedge \Omega\}a(\pi)\{S = true\}$
PPJ.8	If the scenario removeToken is not called by the Owner, it fails	$\{msg.sender \neq p.\_owner\}p.r(\pi)\{S = false\}$
PPJ.9	The removal fails if the token is not in the pool, or if the pool still holds some amount of that token, or if any amount of that token is marked for redemption (pending)	$\{\Omega \vee \pi.balanceOf(p) > 0 \vee p.poolMap[\pi].valueToRedeem > 0\}p.r(\pi)\{S = false\}$
PPJ.10	Otherwise, the scenario is successful	$\{msg.sender = p.owner \wedge \neg\Omega \wedge$
		$\pi.balanceOf(p) = 0 \wedge p.poolMap[\pi].valueToRedeem = 0\}p.r(\pi)\{S = true\}$
PPJ.21	Pool Keeper assignment is successful when and only	$\{\}\kappa(k)\{S = (msg.sender = p.\_owner \wedge k \neq 0)\}$

	when the sender is the Owner	
	In case of success:	
PPN.1	Token is added	$\frac{\{\}a(\pi)\{S=true\}}{\{\forall x:x=p.pool\}a(\pi)\{p.pool=x+TokenParams(\pi,n(\pi),\pi\in ERC4626)\}}$
PPN.1.1		$\frac{\{\}a(\pi)\{S=true\}}{\{\}a(\pi)\{p.poolMap[\pi]=TokenInfo(\pi\in ERC4626?ERC4626:ERC20,n(\pi),p.pools20.length-1),0,false)\}}$
PPN.3	Token is removed	$\frac{\{\}r(\pi)\{S=true\}}{\{\}r(\pi)\{\pi\notin p.pool\}}$
PPN.3.1		$\frac{\{\}r(\pi)\{S=true\}}{\{\}r(\pi)\{\{p.poolMap[\pi]=TokenInfo(None,0,0,false)\}}$
PPN.7	Pool Keeper is properly assigned	$\frac{\{\}\kappa(k)\{S=true\}}{\{\}\kappa(k)\{p.poolKeeper=k\}}$

Redeem

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, r \subset \aleph 256,\ w = p.redeem, k \in \aleph$
		$\widehat{w} = p.supplyManager.redeem\{value = msg.value\}$
		$\{t, v\} = \hat{w}\{value: \$(msg)\}(p, r)$
		$p.poolMap[t].valueToRedeem \geq v$

WPJ.2	If the request is empty, the scenario fails	$\{r = \varnothing\} w(r) \{S = false\}$
WPJ.2a	If reedem is blocked, the scenario fails	$\{p.isRedeemPaused\} w(r) \{S = false\}$
WPJ.3	If the SMRedeem scenario fails, this scenario fails as well	$\{\} \widehat{w}(p, r) \{S = false\} \Rightarrow \{\} w(r) \{S = false\}$
WPJ.3a	If the token does not exist anymore the scenario must fail	$\{p.poolMap[t].tokenType = None\} w(r) \{S = false\}$
WPJ.3b	If the token is blocked, the scenario fails	$\{p.poolMap[t].isBlocked\} w(r) \{S = false\}$
WPJ.4	Otherwise, the scenario is successful	$\frac{\{\}\widehat{w}(p,r)\{S=true\}}{\{r\neq\varnothing\wedge\neg p,isRedeemPaused\wedge p.poolMap[t].tokenType\neq None\wedge\neg p.poolMap[t].isBlocked\}w(r)\{S=true\}}$
	In case of success:	
WPN.1	In case of success, the value to redeem is updated by the normalized value	$\frac{\{\}w(r)\{S=true\}}{\{\forall x{:}x=p.poolMap[t].valueToRedeem\}w(r)\{p.poolMap[t].valueToRedeem=x-v\}}$

Request Redeem

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, t \in A, v \in \aleph, \rho = p.requestRedeem, k \in \aleph$
		$p.poolMap[t].tokenType = ERC20 \Rightarrow k = v10^{-p.poolMap[t].n}$

		$p.poolMap[t].tokenType = ERC4626 \Rightarrow$
		$k = t.convertToShares\left(v10^{-p.poolMap[t].n}\right)$
RPJ.1	If the scenario is invoked not by SM Request Redeem superscenario, it fails	$\{msg.sender \neq p.supplyManager\}\rho(t,v)\{S = false\}$
RPJ.2	If the pool is unknown the scenario fails	$\{p.poolMap[t].tokenType = None\}\rho(t,v)\{S = false\}$
RPJ.3	Otherwise, the scenario is successful	$\{msg.sender = p.supplyManager \wedge (p.poolMap[t].tokenType \neq None)\}$
		$\rho(t,v)\{S = true\}$
	In case of success:	
RPN.1	Value to redeem is increased by value	$\frac{\{\}\rho(t,v)\{S=true\}}{\{\forall x{:}x=p.poolMap[t].valueToRedeem\}\rho(t,v)\{p.poolMap[t].valueToRedeem\}=x+k\}}$
RPN.2	Denormalized value is returned	$\frac{\{\}\rho(t,v)\{S=true\}}{\rho(t,v)=k}$

Set Agents

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, a \in A, e \in B,\ \alpha = \text{'}setAgent\text{'}, \sigma = p.SUPPLY\_MANAGER$
		$\{\}a.getERC20Token()\{S = true\}$

PAJ.1	If the scenario is not initiated by Supply Manager, it fails	$\{s \neq \sigma\}p.\alpha(a, e)\{S = false\}$
PAJ.2	Otherwise, the scenario is successful	$\{s = \sigma\}p.\alpha(a, e)\{S = true\}$
	In case of success:	
PAN.1	In case of enabling, unlimited allowance is set	$\frac{\{\}p.\alpha(a,e)\{S=true \wedge e\}}{\{\}p.\alpha(a,e)\{a.getERC20Token().allowance(p.a)=2^{256}-1\}}$
PAN.2	In case of disabling, zero allowance is set	$\frac{\{\}p.\alpha(a,e)\{S=true \wedge \neg e\}}{\{\}p.\alpha(a,e)\{a.getERC20Token().allowance(p.a)=0\}}$

Set Whitelist

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, w \in A, a = p.'addInWhiteList', d = p,'deleteFromWhiteList'$
		$o = p.\_owner$
PWJ.1	If the sender is not the owner, the add scenario fails	$\{s \neq o\}a(w)\{S = false\}$
PWJ.2	In case of zero element, the add scenario fails	$\{t = 0\}a(w)\{S = false\}$
PWJ.3	In case of known element, the add scenario fails	$\{p.isInWhiteList(t)\}a(w)\{S = false\}$

PWJ.4	Otherwise, the add scenario is successful	$\{s = o \land t \neq 0 \land \neg p.isInWhiteList(t)\}a(w)\{S = true\}$
PWJ.5	If the sender is not the owner, the remove scenario fails	$\{s \neq o\}d(w)\{S = false\}$
PWJ.6	In case of zero element, the remove scenario fails	$\{t = 0\}d(w)\{S = false\}$
PWJ.7	In case of unknown element, the remove scenario fails	$\{\neg p.isInWhiteList(t)\}d(w)\{S = false\}$
PWJ.8	Otherwise, the remove scenario is successful	$\{s = o \land t \neq 0 \land p.isInWhiteList(t)\}d(w)\{S = true\}$
	In case of success:	
PWN.1	New whitelist element is added	$\frac{\{\}a(w)\{S=true\}}{\{\}a(w)\{p.isInWhiteList(w)\}}$
PWN.2	Whitelist element is removed	$\frac{\{\}d(w)\{S=true\}}{\{\}d(w)\{\neg p.isInWhiteList(w)\}}$

Total Supply

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, t = p.totalSupply$
		$\tau = \tau_{20} + \tau_{4626}$

		$\kappa = \pi \in p.token \wedge p.poolMap[\pi.token].tokenType = ERC20$
		$\lambda = \pi \in p.token \wedge p.poolMap[\pi.token].tokenType = ERC4626$
		$\tau_{20} = \sum_{\lambda} 10^{\pi.n}(max(0, \pi.token.balanceOf(p) - \pi.valueToRedeem))$
		$\tau_{4626} = \sum_{\kappa} 10^{\pi.n} max(0, \pi.token.convertToAssets(\pi.token.balanceOf(p)) - \pi.valueToRedeem)$
PTJ.1	The scenario is always successful	$\{\}t()\{S = true\}$
PTN.1	In case of success, total supply is calculated accordingly	$t() = \tau$

Scenario Guardian

N	Human description	Formal description
		$\forall p \in MoleculaPoolTreasury, g \in A, t \in A, b \in B, c = 'changeGuardian',$
		$p_e = 'pauseExecute', p_r = 'pauseRedeem', p_a = 'pauseAll'$
		$u_e = 'unpauseExecute', u_r = 'unpauseRedeem', u_a = 'unpauseAll'$
		$\beta = 'setBlockToken'$

PGJ.1	ChangeGuardian is successful when and only when it is invoked by the owner and a non-zero guardian address	$\{\}p.c(g)\{S = (msg.sender = p.\_owner \wedge g \neq 0)\}$
PGJ.2	pauseExecute is successful when and only when it is invoked either by the owner or by the guardian	$\{\}p.p_e()\{S = (msg.sender = p.\_owner \vee msg.sender = p.guardian)\}$
PGJ.3	pauseRedeem is successful when and only when it is invoked either by the owner or by the guardian	$\{\}p.p_r()\{S = (msg.sender = p.\_owner \vee msg.sender = p.guardian)\}$
PGJ.4	pauseAll is successful when and only when it is invoked either by the owner or by the guardian	$\{\}p.p_a()\{S = (msg.sender = p.\_owner \vee msg.sender = p.guardian)\}$
PGJ.5	unpauseExecute is successful when and only when it is invoked by the owner	$\{\}p.u_e()\{S = (msg.sender = p.\_owner)\}$
PGJ.6	unpauseRedeem is successful when and only when it is invoked by the owner	$\{\}p.u_r()\{S = (msg.sender = p.\_owner)\}$
PGJ.7	unpauseAll is successful when and only when it is invoked by the owner	$\{\}p.u_a()\{S = (msg.sender = p.\_owner)\}$
PGJ.8	setBlockToken is successful	$\{\}p.\beta(t, b)\{S = (msg.sender = p.\_owner \wedge p.poomMap[t].tokenType \neq None)\}$

	when and only when it is invoked by the owner and token exists in the pool	
	In case of success:	
PGN.1	In case changeGuardian is successful, the guardian is changed accordingly	$\frac{\{\}p.c(g)\{S=true\}}{\{\}p.c(g)\{ap.guardian=g\}}$
PGN.2	In case pauseExecute is successful, the execute is paused	$\frac{\{\}p.p_e()\{S=true\}}{\{\}p.p_e\{p.isExecutePaused\}}$
PGN.3	In case pauseRedeem is successful, the redeem is paused	$\frac{\{\}p.p_r()\{S=true\}}{\{\}p.p_r\{p.isRedeemPaused\}}$
PGN.4	In case pauseAll is successful, the redeem and execute are paused	$\frac{\{\}[.p_a()\{S=true\}}{\{\}p.p_a\{a.isRedeemPaused \wedge a.isExecutePaused\}}$
PGN.5	In case unpauseExecute is successful, the execute is unpaused	$\frac{\{\}p.u_e()\{S=true\}}{\{\}p.u_e\{\neg a.isExecutePaused\}}$
PGN.6	In case unpauseRedeem is successful, the redeem is unpaused	$\frac{\{\}p.u_{r_r}()\{S=true\}}{\{\}p.u_r\{\neg a.isRedeemPaused\}}$
PGN.7	In case unpauseAll is successful, the redeem and execute are unpaused	$\frac{\{\}p.u_a()\{S=true\}}{\{\}p.u_a\{\neg(p.isRedeemPaused \vee p.isExecutePaused)\}}$

PGN.8	In case setBlockToken is successful, the token is correspondingly blocked or unblocked	$\frac{\{\}p.\beta(t,b)\{S=true\}}{\{\}p.\beta(t,b)\{p.poolMap[t].isBlocked\}=b}$

Nitrogen

Deposit

N	Human description	Formal description
		$\forall a \in AccountantAgent, i \in \aleph256, u \in A, v \in \aleph256, r = 'requestDeposit'$
		$\hat{\$} = a.ERC20\_TOKEN,\ \sigma = r.SUPPLY\_MANAGER,\ \tau = a.rebaseToken$
		$\kappa = (msg.sender = \tau) \land (\{\}\ \hat{\$}.safeTransferFrom(u, a, v)\{S = true\}) \land$
		$(\$(msg) = 0) \land \neg a.isRequestDepositPaused$
		$d = \sigma.deposit,\ \varsigma = d(\hat{\$}, i, v), c = \tau.confirmDeposit$
		$\zeta = (\{\}d(\hat{\$}, i, v)\{S = true\}) \land (\{\}c(i, \varsigma)\{S = true\})$
DNI.1	In case of wrong sender, the scenario fails	$\{msg.sender \neq \tau\}a.r(i, u, v)\{S = false\}$
DNI.2	In case of failed transfer, the scenario fails	$\{\}\ \hat{\$}.safeTransferFrom(u, a, v)\{S = false\} \Rightarrow \{\}a.r(i, u, v)\{S = false\}$

DNI.3	In case of non-zero value, the scenario fails	$\{\$(msg) > 0\}a.r(i,u,v)\{S = false\}$
DNI.3a	In case the depositing paused, the scenario fails	$\{a.isRequestDepositPaused\}a.r(i,u,v)\{S = false\}$
DNI.4	Otherwise, the scenario is successful if and only if, underlying scenarios are successful	$\frac{\zeta}{\{\kappa\}a.r(i,u,v)\{S=true\}}$
	In case of success:	
DNO.1	Prepare block increases balance	$\frac{\{\}a.r(i,u,v)\{S=true\}}{\{\}a.r(i,u,v)\{\uparrow\hat{\$}.safeTransferFrom(u,a,v)\}}$
DNO.2	Prepare block sets allowance	$\frac{\{\}a.r(i,u,v)\{S=true\}}{\{\}a.r(i,u,v)\{\uparrow\hat{\$}.forceApprove(\sigma.getMoleculaPool(),v)\}}$
DNO.3	The scenario is a superposition of underlying blocks and scenarios	$\frac{\{\}a.r(i,u,v)\{S=true\}}{\{\}a.r(i,u,v)\{\uparrow d(\hat{\$},i,v)\}}$
DNO.3.1		$\frac{\{\}a.r(i,u,v)\{S=true\}}{\{\}a.r(i,u,v)\{\uparrow c(i,\varsigma)\}}$

Redeem

Scenario Nitrogen Redeem Request

N	Human description	Formal description

		$\forall a \in AccountantAgent, i \in \aleph256, v \in \aleph256, r = \text{'}requestRedeem\text{'}, \tau = a.rebaseToken$
RNI.1	If the scenario is not initiated by the proper superscenario, it fails	$\{msg.sender \neq \tau\}a.r(i,v)\{S = false\}$
RNI.2	If the message value is positive,the scenario fails	$\{\$(msg) > 0\}a.r(i,v)\{S = false\}$
RNI.2a	If the redeeming is paused, the scenario fails	$\{a.isRequestRedeemPaused\}a.r(i,v)\{S = false\}$
RNI.3	Otherwise, the scenario is successful if and only if the underlying scenario is successful	$\frac{\{\}\tau.SUPPLY\_MANAGER.requestRedeem(ERC20\_TOKEN,i,v)\{S=true\}}{\{msg.sender=\tau \land \$(msg)=0 \land \neg a.isRequestRedeemPaused\}a.r(i,v)\{S=true\}}$
	In case of success:	
RNO.1	The result of the scenario is a result of the underlying MoleculaPool scenario	$\frac{\{\}a.r(i,v)\{S=true\}}{\{\}a.r(i,v)\{\uparrow\tau.SUPPLY\_MANAGER.requestRedeem(ERC20\_TOKEN,i,v)\}}$

Scenario Nitrogen Redeem

N	Human description	Formal description
		$\forall a \in AccountantAgent, k \in A, \rho \in [\aleph256], v \in [\aleph256], t \in \aleph256,\ r = \text{'}redeem\text{'}$
		$\sigma = a.SUPPLY\_MANAGER, \tau = a.rebaseToken, \hat{\$} = a.ERC20\_TOKEN$

		$\kappa = (msg.sender = \sigma) \land (\$(msg) = 0) \land$
		$\{\}\ \hat{\$}.safeTransferFrom(k, a, t)\{S = true\}$
WNI.1	If the scenario is initiated not by 'SM Redeem', it fails	$\{msg.sender \neq \sigma\}a.r(k, \rho, v, t)\{S = false\}$
WNI.2	If the message value is not zero, the scenario fails	$\{\$(msg) > 0\}a.r(k, \rho, v, t)\{S = false\}$
WNI.3	If the keeper balance is too low, the scenario fails	$\{\}\ \hat{\$}.safeTransferFrom(k, a, t)\{S = false\} \Rightarrow \{\}a.r(k, \rho, v, t)\{S = false\}$
WNI.5	Otherwise, the scenario is successful if and only if underlying scenario is successful	$\frac{\{\}\tau.redeem(\rho,v)\{S=true\}}{\{\kappa\}a.r(k,\rho,v,t)\{S=true\}}$
	In case of success:	
WNO.1	Transfer increases accountant balance by the required amount	$\frac{\{\}a.r(k,\rho,v,t)\{S=true\}}{\{\}a.r(k,\rho,v,t)\{\uparrow\hat{\$}.safeTransferFrom(k,a,t)\}}$
WNO.2	Transfer decreases accountant balance by the required amount	Covered by WNO.1
WNO.3	In case of success the scenario is a composition of the Transfer and the underlying scenario	$\frac{\{\}a.r(k,\rho,v,t)\{S=true\}}{\{\}a.r(k,\rho,v,t)\{\uparrow\tau.redeem(\rho,v)\}}$

Scenario Nitrogen Redeem Confirm

N	Human description	Formal description
		$\forall a \in AccountantAgent, u \in \aleph 256, v \in \aleph 256,\ r = 'confirmRedeem'$
		$\hat{\$} = a.ERC20\_TOKEN$
ONI.1	If the scenario is not initiated by 'Token Redeem Confirm' superscenario, it fails	$\{msg.sender \neq a.rebaseToken\}r(u, v)\{S = false\}$
ONI.2	If there are too few assets, the scenario fails	$\{\hat{\$}.balanceOf(a) < v\}r(u, v)\{S = false\}$
ONI.3	Otherwise, the scenario is successful	$\{msg.sender = a.rebaseToken \wedge \hat{\$}.balanceOf(a) \geq v\}r(u, v)\{S = true\}$
	In case of success:	
ONO.1	Balance of the controller is increased accordingly	$\frac{\{\}r(u,v)\{S=true\}}{\{\forall x: x=\hat{\$}.balanceOf(u) \wedge a \neq u\}r(u,v)\{\hat{\$}.balanceOf(u)=x+v\}}$
ONO.2	Balance of accountant is decreased accordingly	$\frac{\{\}r(u,v)\{S=true\}}{\{\forall x: x=\hat{\$}.balanceOf(a) \wedge a \neq u\}r(u,v)\{\hat{\$}.balanceOf(a)=x-v\}}$

Scenario Nitrogen Distribute

N	Human description	Formal description

		$\forall a \in AccountantAgent, u \in [\aleph 256], v \in [\aleph 256],\ d = 'distribute'$
		$\tau = a.rebaseToken,\ i \in [0..u.length - 1], u.length = v.length,$
		$\kappa = \forall x \in u{:}\ x \neq 0$
BNI.1	If the scenario is not initiated by 'Distribute Yield', it fails	$\{msg.sender \neq a.SUPPLY\_MANAGER\}a.d(u,v)\{S = false\}$
BNI.2	If the message value is not zero , the scenario fails	$\{\$(msg) > 0\}a.d(u,v)\{S = false\}$
BNI.2.1	If any of the users is zero the scenario fails	$\{\neg\kappa\}a.d(u,v)\{S = false\}$
BNI.3	Otherwise, the scenario is successful	$\{msg.sender = a.SUPPLY\_MANAGER \wedge \$(msg) = 0 \wedge \kappa\}a.d(u,v)\{S = true\}$
	In case of success:	
BNO.1	For each user the corresponding amount of shares is provided	$\frac{\{\}a.d(u,v)\{S=true\}}{\{\forall x{:}x=\tau.sharesOf(u[i])\}a.d(u,v)\{\tau.sharesOf(u[i])=x+v[i]\}}$

Scenario Nitrogen Guardian

N	Human description	Formal description
		$\forall a \in AccountantAgent, g \in A,\ c = 'changeGuardian',$
		$p_d = 'pauseRequestDeposit',\ p_r = 'pauseRequestRedeem', p_a = 'pauseAll'$

		u_d = 'unpauseRequestDeposit', u_r = 'inpauseRequestRedeem', u_a = 'unpauseAll'
GNI.1	ChangeGuardian is successful when and only when it is invoked by the owner and a non-zero value	$\{\}a.c(g)\{S = (msg.sender = a.\_owner \wedge g \neq 0)\}$
GNI.2	pauseDeposit is successful when and only when it is invoked either by the owner or by the guardian	$\{\}a.p_d()\{S = (msg.sender = a.\_owner \vee msg.sender = a.guardian)\}$
GNI.3	pauseRedeem is successful when and only when it is invoked either by the owner or by the guardian	$\{\}a.p_r()\{S = (msg.sender = a.\_owner \vee msg.sender = a.guardian)\}$
GNI.4	pauseAll is successful when and only when it is invoked either by the owner or by the guardian	$\{\}a.p_a()\{S = (msg.sender = a.\_owner \vee msg.sender = a.guardian)\}$
GNI.5	unpauseDeposit is successful when and only when it is invoked by the owner	$\{\}a.u_d()\{S = (msg.sender = a.\_owner)\}$
GNI.6	unpauseRedeem is successful when and only when it is invoked by the owner	$\{\}a.u_r()\{S = (msg.sender = a.\_owner)\}$
GNI.7	unpauseAll is successful when and only when it is invoked by	$\{\}a.u_a()\{S = (msg.sender = a.\_owner)\}$

	the owner	
	In case of success:	
GNO.1	In case changeGuardian is successful, the guardian is changed accordingly	$\frac{\{\}a.c(g)\{S=true\}}{\{\}a.c(g)\{a.guardian=g\}}$
GNO.2	In case pauseDeposit is successful, the deposit is paused	$\frac{\{\}a.p_d()\{S=true\}}{\{\}a.p_d\{a.isRequestDepositPaused\}}$
GNO.3	In case pauseRedeem is successful, the redeem is paused	$\frac{\{\}a.p_r()\{S=true\}}{\{\}a.p_r\{a.isRequestRedeemPaused\}}$
GNO.4	In case pauseAll is successful, the redeem and deposit are paused	$\frac{\{\}a.p_a()\{S=true\}}{\{\}a.p_a\{a.isRequestRedeemPaused \land a.isRequestDepositPaused\}}$
GNO.5	In case unpauseDeposit is successful, the deposit is unpaused	$\frac{\{\}a.u_d()\{S=true\}}{\{\}a.u_d\{\neg a.isRequestDepositPaused\}}$
GNO.6	In case unpauseRedeem is successful, the redeem is unpaused	$\frac{\{\}a.u_r()\{S=true\}}{\{\}a.u_r\{\neg a.isRequestRedeemPaused\}}$
GNO.7	In case unpauseAll is successful, the redeem and deposit are unpaused	$\frac{\{\}a.u_a()\{S=true\}}{\{\}a.u_a\{\neg(a.isRequestRedeemPaused \lor a.isRequestDepositPaused)\}}$

Verification

The verification code is placed here. 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 A.

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.
- Major - possibility for incorrect system behavior preventing the user from performing their goals, but without losing their assets.
- Minor - any minor issues
- Waived - all the issues that were waived
- Fixed - all the fixed issues

IMPORTANT NOTICE!!! All the fixed or waived issues are marked as Minor or Fixed!!!

List of found issues

N	Description	Status
Issues as of 01/10/25		
1	*MoleculaPool*: In some cases such as: - all the pools empty - no pools - all the values are to be redeem can return zero value that will lead to division by zero. After discussions agreed that such cases are practically impossible	MINOR
2	*MoleculaPool:* No check for duplicated pools. Lack of such check will lead to incorrect calculation of total supply.	FIXED
3	*MoleculaPool:* In case index *i* is out of bounds, the setPool20 method will quietly do nothing, that will prevent the operator to know that something went wrong.	FIXED
4	*MoleculaPool:* In case a pool with some value is popped, total supply will become incorrect (redeem would not be possible as well).	FIXED
5	*MoleculaPool:* No support for depositing or redeeming ERC4626 tokens.	FIXED
6	*SupplyManager:* *apyFormatter* should not be zero, otherwise division by zero will occur during the calculation of *operationYield* in *requestRedeem*	FIXED
8	*SupplyManager:* no check if the party's agent is registered as an agent for yield distribution	FIXED
9	*SupplyManager:* No check if there is a positive income in case of yield distribution	FIXED

Issues as of 01/21/25		
10	*RebaseToken:* Minimal values must be positive	FIXED
11	*RebaseToken:* It's possible to perform very small redeem requests	FIXED
Issues as of 02/06/25		
12	*MoleculaPool:* No check for zero balance of old token in case of *setToken* No practical meaning, as the method is deprecated	WAIVED
Issues as of 03		
13	*MoleculaPool*: In case of migration zero value to redeem in the new pool assumed No practical meaning, as such an assumption confirmed by the developers as a correct one, it's planned to use a "clean" new *Molecula Pool* before migration	WAIVED
14	*MoleculaPool*: No check for zero balance for *usdtAddress* and *guardianAddress* in the *constructor*	FIXED

Appendix A: 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.

Briefly the process of the formal verification looks as follows:

1. *Ursus* software must be installed as described above
2. Coq level specification must be created from the Low-level specification
3. Source code must be translated into Ursus DSL
4. Then, Ursus is executed against the specification and the code and tries to prove that the code satisfies the specification
5. Execution takes some time (usually, from 10 minutes to a few days) and takes a huge amount of RAM (32 Gb RAM and 96 Gb of swap) are required
6. If *Ursus* manages to prove that the code is correct against the specification it returns a positive result (**OK**)
7. Otherwise, Ursus returns the negative result (error message). In this case, the verifier analyzes the failure. The possible causes are as follows:
    a. Implementation error. In this case the developer is informed, the bug is fixed (the process is returned to step 2), or not fixed. In the latter case, the specification is modified and the process is returned to step 3
    b. Specification error. In this case, the specification is modified and the process is returned to step 3
    c. Some statements are too difficult for *Ursus*. In this case, the special hints and rewording is used, the process is repeated from the step 3, again and again, until the result is reached

When the final result is reached, the execution of *Ursus* just returns a positive result.

Technical instructions are described below.

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. Install *coq*:
    a. *sudo apt install opam*
    b. *opam init*
    c. *eval $(opam env)*
    d. *opam pin add coq 8.16.1*
    e. *opam repo add coq-released https://coq.inria.fr/opam/released*
4. 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/coq-tvm-model
   d. https://vcs.modus-ponens.com/ton/solidity-monadic-language
   e. https://vcs.modus-ponens.com/ton/pruvendo-base-lib
   f. https://vcs.modus-ponens.com/ton/ursus-standard-library
   g. https://vcs.modus-ponens.com/ton/pruvendo-ursus-tvm
   h. https://vcs.modus-ponens.com/ton/ursus-contract-creator
   i. https://vcs.modus-ponens.com/ton/ursus-proofs
   j. https://vcs.modus-ponens.com/ton/ursus-environment
   k. https://vcs.modus-ponens.com/ton/ursus-quickchick
   l. https://vcs.modus-ponens.com/ton/coq-tvm-model.git

5. 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 coq-tvm-model && git checkout tvmcells
opam install --ignore-pin-depends -y .
cd ..
cd solidity-monadic-language && git checkout new_break
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 new_assign
opam install --ignore-pin-depends -y .
cd ..
cd pruvendo-ursus-tvm && git checkout new_assign
opam install --ignore-pin-depends -y .
cd ..
cd ursus-contract-creator && git checkout new_assign
opam install --ignore-pin-depends -y .
cd ..
cd ursus-proofs && git checkout new_return
opam install --ignore-pin-depends -y .
cd ..
cd ursus-environment && git checkout new_cells
opam install --ignore-pin-depends -y .
cd ..
cd ursus-quickchick && git checkout new_return
opam install --ignore-pin-depends -y .
cd ..
cd ursus-lib-execs && git checkout new_structure_exit
opam install --ignore-pin-depends -y .
```

6. Clone the project repository mentioned above
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, no errors occurred