Formal Verification Engagement Summary: Term Finance

Introduction

In September 2024, Runtime Verification engaged with Term Finance to perform formal verification on their DeFi protocol. At Runtime Verification, we specialize in building rigorous, mathematically grounded proofs to ensure that smart contracts behave as intended under all possible inputs and scenarios. Our tool, **Kontrol**, is designed to integrate seamlessly with Solidity-based projects, enabling developers to write property-based tests in Solidity and leverage symbolic execution to verify them. The following document summarizes the key aspects of our formal verification engagement with Term Finance, focusing on the tests, invariants, and findings from this collaboration.

Reproducing the Proofs

To reproduce the results of this verification locally, follow the steps below:

Install Kontrol:

```
bash <(curl https://kframework.org/install)
kup install kontrol
```

Clone the Test Repository:

```
git clone git@github.com:term-finance/yearn-v3-term-vault.git
cd yearn-v3-term-vault.git
git submodule update --init --recursive
```

Run the Tests:

```
kontrol build
kontrol prove --config-profile setup
kontrol prove --config-profile tests
```

View the Results:

`kontrol list` (optionally, it is possible to pass the flag `--xml-test-report` to the `kontrol prove` command and obtain a report in XML format)

Scope

The goal of this engagement was to verify the correctness of two implementations of linked lists in Solidity, and more concretely, verify that the insertion and removal algorithms of both implementations preserved desired properties.

The files under the scope of the engagement were:

- `RepoTokenList.sol`
- `RepoTokenUtils.sol`
- `TermAuctionList.sol`

We used the branch `runtime-fv` to sync the changes and fixes implemented by the Term team and to push our tests and improvements to the code. The initial commit we started working on was `f593f028be99e2dc7e0a210555130511b8fd61db`. We ran the tests after changes and fixes to the code to ensure that those changes wouldn't break the properties and the fixes were actually fixing the found issue. The last commit we ran all the property tests is `6b06e5e73e21f34acd313ae98668d0fed0816762` where all proofs were passing.

Invariants Proven

Below are the definitions of both linked lists and the invariants we sought to formally verify in this engagement:

RepoTokenList

The `head` of `RepoTokenListData` contains either `NULL_NODE` (equal to `bytes32(0)`) or the address of a RepoToken (whose contract can be found here). The `nodes` mapping should store the `RepoTokenListNode` with the `next` `RepoToken` in the list for each `RepoToken`.
The `discountRates` mapping in `RepoTokenListData` stores the discount rate of each `repoToken` in the list. The `collateralTokenParams` stores the minimum collateral ratio of each collateral token.

```
struct RepoTokenListNode {
    address next;
}

struct RepoTokenListData {
    address head;
    mapping(address => RepoTokenListNode) nodes;
    mapping(address => uint256) discountRates;
    /// @notice keyed by collateral token
    mapping(address => uint256) collateralTokenParams;
}
```

RepoTokenList Invariants:

- **Invariant 1:** Unique Entries
    - Status: Verified
    - Summary: There should never be duplicate `repoToken` addresses in the list.

- **Invariant 2:** List Integrity
  - Status: Verified
  - Summary: The number of nodes counted by traversing the list should always equal the number of unique `repoTokens` stored.
  - Observation: We check that the list count remains consistent after each operation. The Unique Entries check ensures that all `RepoTokens` are unique. Together, these two prove that the invariant holds.

- **Invariant 3:** Null Termination
  - Status: Verified
  - Summary: The `next` pointer of the last node in the list should always be `NULL_NODE`.
  - Observation: We check that the list count remains consistent after each operation. If the proof terminates, the list is guaranteed to be null-terminated.

- **Invariant 4:** Head Validity
  - Status: Verified
  - Summary: If the list is non-empty, the `head` should never be `NULL_NODE`; if the list is empty, the `head` should always be `NULL_NODE`.
  - Observation: We check that the list count remains consistent after each operation. The count returns 0 if, and only if, the head is null.

- **Invariant 5:** Sorted by Maturity
  - Status: Verified
  - Summary: The `repoTokens` in the `RepoTokenList` should always be sorted by maturity.

- **Invariant 6:** Insert Operation Correctness
  - Status: Verified
  - Summary: After inserting a new `repoToken`, it should be in the correct sorted position based on its maturity.
  - Observation: We check that the inserted token is in the list, and we check that the original list is preserved, i.e. all elements that were in the list before are still in the list, and we also check that the list is sorted by maturity after the insertion.

- **Invariant 7:** Maturity Validity
  - Status: Verified
  - Summary: All `repoTokens` in the list should have a maturity `timestamp` greater than the current block `timestamp`.

- **Invariant 8:** Balance Consistency
  - Status: Verified
  - Summary: The `Strategy`'s balance for each repoToken in the list should be greater than zero.

- **Invariant 9:** Remove Operation Correctness
  - Status: Verified
  - Summary: After removing a `repoToken`, it should no longer be in the list, and the list should maintain its sorted order.
  - Observation: The checks for Maturity Validity, Sorted by maturity and Balance Consistency ensure that all matured and redeemed tokens have been removed and the list remains sorted. Additionally, we check that all the elements that are in the resulting list after remove were in the list before.

TermAuctionList

Each `PendingOffer` stores information about an offer for some auction. The `repoToken` is an `ERC20` token whose implementation can be found here. The `termAuction` is the auction contract address that the `offerAmount` refers to. Finally, the offerLocker is the contract that handles and processes auction offer submissions.

```
struct PendingOffer {
    address repoToken;
    uint256 offerAmount;
    ITermAuction termAuction;
    ITermAuctionOfferLocker offerLocker;
}
```

The `head` of `TermAuctionListData` contains either `NULL_NODE` (equal to `bytes32(0)`) or some `offerId`, which which is generated using a hash function. The `nodes` mapping should store the `TermAuctionList` node with the `next offerId` for each `offerId`. The `offers` mapping should store the corresponding `PendingOffer` for each `offerId`.

```
struct TermAuctionListNode {
    bytes32 next;
}

struct TermAuctionListData {
    bytes32 head;
    mapping(bytes32 => TermAuctionListNode) nodes;
    mapping(bytes32 => PendingOffer) offers;
}
```

TermAuctionList Invariants:

- **Invariant 1:** Unique Entries
  - Status: Verified
  - Summary: This invariant ensures that there should never be duplicate offer IDs in the list.

- **Invariant 2:** List Integrity
  - Status: Verified
  - Summary: This invariant ensures that the number of nodes counted by traversing the list should always equal the number of unique offer IDs stored.
  - Observation: We check that the list count remains consistent after each operation. The Unique Entries check ensures that all offer IDs are unique. Together, these two prove that the invariant holds.

- **Invariant 3:** Null Termination
  - Status: Verified
  - Summary: This invariant ensures that the `next` pointer of the last node in the list should always be `NULL_NODE`.
  - Observation: We check that the list count remains consistent. If the proof terminates, the list is guaranteed to be null-terminated.

- **Invariant 4:** Head Validity
  - Status: Verified
  - Summary: This invariant ensures that if the list is non-empty, the `head` should never be `NULL_NODE`; if the list is empty, the `head` should always be `NULL_NODE`.
  - Observation: We check that the list count remains consistent after each operation. The count returns 0 if, and only if, the head is null.

- **Invariant 5:** Sorted by Auction Address
  - Status: Verified
  - Summary: This invariant ensures that the `offers` in the `TermAuctionList` should always be sorted by the `termAuction` address.

- **Invariant 6:** Insert Operation Correctness
  - Status: Verified
  - Summary: This invariant ensures that after inserting a new `offer`, it should be in the correct sorted position based on its `termAuction` address.
  - Observation: We check that the inserted new `offer` is in the list, and the check for Sorted by auction address invariant ensures that it is in the correct position. Together, these two prove that the invariant holds.

- **Invariant 7:** Auction Status Consistency
  - Status: Verified
  - Summary: This invariant ensures that there should be no offers in the list for completed or canceled auctions.

- **Invariant 8:** Positive Offers Consistency
  - Status: Verified
  - Summary: This invariant ensures that all offers in the list have a positive balance.

- **Invariant 9:** Remove Operation Correctness
  - Status: Verified
  - Summary: This invariant ensures that after removing a completed or canceled offer, it should no longer be on the list, and the list should maintain its sorted order.

    - Observation: The checks for Sorted by auction address and Auction Status Consistency ensure that all offers for completed or canceled auctions have been removed and that the list remains sorted.

- **Invariant 10:** Offer Consistency
    - Status: Verified
    - Summary: This invariant ensures that the offer amount stored in the list for each offer should match the amount locked in the corresponding `TermAuctionOfferLocker` contract.

- **Invariant 11:** Valid RepoTokens Consistency
    - Status: Verified
    - Summary: This invariant ensures that, for any `offer` in the list its corresponding `repoToken` is valid according to the checks performed by the function `validateRepoToken`

Proof Overview

The invariant proofs of the `RepoTokenListData` can be found in the `RepoTokenListInvariants` contract, whereas the invariant proofs for the `TermAuctionListData` can be found in the `TermAuctionListInvariants` contract. Both contracts inherit from `KontrolTests` contract, which inherits the Foundry `Tests` contract and also the `KontrolCheats` contract. The `KontrolCheats` contains the `kevm` address as well as the library of Kontrol cheatcodes that are necessary to run the tests.

Both `RepoTokenListInvariants` and `TermAuctionListInvariants` have a setUp method that makes their respective storage all symbolic through the Kontrol cheatcode `kevm.symbolicStorage(address(this))`. The use of this cheatcode in `setUp` makes the storage of respective contracts symbolic, or arbitrary, which allows proving that invariants hold for any state of the contract and its storage.

After setting the storage to symbolic, the `setUp` function applies loop unrolling via the `bmc-depth` in the `kontrol.toml` file, creating arbitrary lists with sizes 0, 1, …, `bmc-depth`. This parameter can be passed in the `cli`, overriding the one in the `toml` file. The proofs therefore offer a **bounded verification guarantee**, for lists of sizes up to and including bmc-depth.

We designed the following tests to prove the specified invariants:

RepoTokenList

- ☑ **Test 1: Property-Based Test for Insert a New RepoToken in the List**
    - **Objective:** This test verifies that `insertSorted` inserts a new `repoToken` in the list (that was not present in the list before) while preserving the list invariants.
    - **Current Status:** Verified and passed in **6m:31s** for bmc-depth 3.
    - **Details:** Before inserting a new token, we assume that the invariants are holding. Then, we build a new repo token, which we know is not in the list because it is the result of `_newRepoToken()`. Finally, we assert that the repo token is in the list and that the invariants are still holding.
    - **Assumptions before function call:**
        - There are no matured tokens in the list
        - The tokens in the list are sorted by maturity
        - All tokens in the list have a positive balance
        - The new token to be inserted has not matured yet and has a positive balance

- **Assertions:**
  - **Before function call:**
    - There are no duplicate tokens in the list - this is asserted instead of assumed since the initialization procedure guarantees this invariant
  - **After function call:**
    - The new token is in the list
    - The length of the list is equal to the previous length plus one
    - There are no duplicate tokens in the list
    - There are no matured tokens in the list
    - The tokens in the list are sorted by maturity
    - All tokens in the list have a positive balance

☑ **Test 2: Property-Based Test for Inserting a Duplicated RepoToken in the List**

- **Objective:** This test verifies that `insertSorted` preserves the list invariants when trying to insert a RepoToken that is already in the list.

- **Current Status:** Verified and passed in **4m:16s** for bmc-depth 3.

- **Details:** We assume that the invariants are holding. Then, we try to insert a repo token that is already in the list. Finally, we assert that the invariants are still holding and that the size of the list didn't change.

- **Assumptions before function call:**
  - There are no matured tokens in the list
  - The tokens in the list are sorted by maturity
  - All tokens in the list have a positive balance

- **Assertions:**
  - **Before function call:**
    - There are no duplicate tokens in the list - this is asserted instead of assumed since the initialization procedure guarantees this invariant
  - **After function call:**
    - The length of the list is equal to the previous length
    - There are no duplicate tokens in the list
    - There are no matured tokens in the list
    - The tokens in the list are sorted by maturity
    - All tokens in the list have a positive balance

☑ **Test 3: Property-Based Test for Removing Matured Tokens from the List**

- **Objective:** This test verifies that `removeAndRedeemMaturedTokens` removes all the matured tokens from the list, preserving the list invariants
- **Current Status:** Verified and passed in **5m:6s** for bmc-depth 3.
- **Details:** We assume that desired invariants hold. We cannot assume that there are no matured tokens in the list or that all tokens in the list have a positive balance because the purpose of the `removeAndRedeemMaturedTokens` is to remove these tokens from the list. We will instead assert that those invariants hold after calling `removeAndRedeemMaturedTokens`, together with the other invariants that were holding before.
- **Assumptions:**
  - The tokens in the list are sorted by maturity
  - The call to `redeemTermRepoTokens` will not revert
- **Assertions:**
  - **Before function call:**
    - There are no duplicate tokens in the list - this is asserted instead of assumed since the initialization procedure guarantees this invariant
  - **After function call:**
    - The length of the list is less than or equal to the previous length
    - There are no duplicate tokens in the list
    - The tokens in the list are sorted by maturity
    - There are no matured tokens in the list
    - All tokens in the list have a positive balance

TermAuctionList

- ☑ **Test 1: Property-Based Test for Inserting a New Offer in the List**
    - **Objective:** This test verifies that `insertPending` inserts a new offer in the list (that was not present in the list before) and preserves the list invariants.
    - **Current Status:** Verified and passed in **23m:31s** for bmc-depth 3.
    - **Details:** Before inserting a new offer in the list, we assume that the invariants are holding. The test receives as a parameter an `offerId` that we assume is not on the list yet. Then we build a new offer to be submitted together with this `offerId`. Finally, we assert that the new offer is on the list and that the invariants are still holding.
    - **Assumptions before function call:**
        - The new `offerId` to be inserted is not on the list
        - For any offer in the list, the offer amount matches the locked offer amount in the respective locker
        - For any offer in the list, the offer amount is positive
        - There are no completed or canceled auctions on the list
        - The `termAuction` of the offer is not completed
        - The `termAuction` of the offer to be inserted is not cancelled for withdrawal
        - The new `offerId` to be inserted is different than `bytes32(0)`
        - The new offer amount to be inserted is equal to the locked offer amount
        - The locked offer amount for the offer to be inserted is greater than 0
        - The `offer.repoToken` is a valid `repoToken`, i.e. it passes the checks performed by `validateRepoToken` function
    - **Assertions:**
        - **Before function call:** initialization procedure guarantees these invariants
            - The list is sorted by auction address
            - There are no duplicated offers in the list
        - **After function call:**
            - The new offer is on the list
            - The length of the list is equal to the previous length plus one
            - There are no duplicated offers in the list
            - The offers in the list are sorted by auction address
            - For any offer in the list, the offer amount matches the locked offer amount in the respective locker
            - For any offer in the list, the offer amount is positive
            - There are no completed or canceled auctions on the list

☑ **Test 2: Property-Based Test for Inserting a Duplicated Offer in the List**

- **Objective:** This test verifies that `insertPending` preserves the list invariants when trying to insert an offer whose `offerId` is already in the list.

- **Current Status:** Verified and passed in **11m:03s** for bmc-depth 3.

- **Details:** We assume that the invariants are holding. Then, we insert an `offerId` that is already in the list. Finally, we assert that the offer amount was updated, the invariants are still holding, and that the size of the list didn't change.

- **Assumptions before function call:**
  - The `offerId` to be inserted is already on the list
  - For any offer in the list that is different from the `offerId` to be inserted, the offer amount matches the locked offer amount in the respective locker. The point of calling `insertPending` with an offer that is already on the list is to update the offer amount
  - For any offer in the list, the offer amount is positive
  - There are no completed or canceled auctions on the list
  - The `termAuction` of the `pendingOffer` to be submitted should be equal to the `termAuction` with the same `offerId` that is already on the list
  - The `repoToken` of the `pendingOffer` to be submitted should be equal to the `repoToken` with the same `offerId` that is already on the list
  - The `offerLocker` of the `pendingOffer` to be submitted should be equal to the `offerLocker` with the same `offerId` that is already on the list
  - The offer amount of the `pendingOffer` to be submitted is greater than 0
  - The offer amount of the `pendingOffer` to be submitted is equal to the locked offer amount

- **Assertions:**
  - **Before function call:** initialization procedure guarantees these invariants
    - The list is sorted by auction address
    - There are no duplicated offers in the list
  - **After function call:**
    - The offer is on the list and the offer amount has been updated to the new amount
    - The length of the list is equal to the previous length plus one
    - There are no duplicated offers in the list
    - The offers in the list are sorted by auction address
    - For any offer in the list, the offer amount matches the locked offer amount in the respective locker
    - For any offer in the list, the offer amount is positive
    - There are no completed or canceled auctions on the list

☑ **Test 3: Property-Based Test for Remove Offers for Completed or Cancelled auctions from the list**

- **Objective:** This test verifies that the `removeCompleted` function removes all offers for auctions that have been cancelled or completed, or offers whose locked offer amount is 0 and preserves the list invariants.

- **Current Status:** Verified and passed in **1h:19m:12s** for bmc-depth 3.

- **Details:** We assume that desired invariants hold. We cannot assume that there are no offers for completed or canceled auctions or that the offer amount is equal to the locked offer amount for each offer because the purpose of the `removeCompleted` function is to remove all these offers from the list. We will instead assert that, after calling `removeCompleted`, there are no offers for completed or canceled auctions in the list and all offers have a positive locked offer amount, together with the other invariants that were holding before.

- **Assumptions:**
  - For any offer in the list, the offer amount is positive
  - The calls to the `offerLocker.unlockOffers` will always succeed
  - The calls to the `validateAndInsertRepoToken` will not revert. This assumption is for proof efficiency, otherwise, the proof would branch in every condition of `validateAndInsertRepoToken` making it very hard to prove for bmc depths greater than 3
  - For proof simplicity: the `RepoTokenList` is empty, and there are no discount rates set for any of the list offers repoTokens

- **Assertions:**
  - **Before function call:** initialization procedure guarantees these invariants
    - The list is sorted by auction address
    - There are no duplicated offers in the list
  - **After function call:**
    - The length of the list is less or equal to the previous length
    - There are no duplicated offers in the list
    - The offers in the list are sorted by auction address
    - For any offer in the list, the offer amount is positive
    - There are no completed or canceled auctions on the list
    - For any offer in the list, the locked offer amount is positive

Get Functions Properties

After proving the correctness of both lists, the client wanted to verify the properties of their `get` functions. Both lists had implementations for `getPresentValue` and `getCumulativeData` (`getCumulativeRepoTokenData` and `getCumulativeOfferData` for `RepoTokenList` and `TermAuctionList` respectively).

The properties asked to be verified are the following:

- **Property 1:** Cumulative Offer Data Consistency
    - The cumulative weighted time to maturity and cumulative offer amount calculated by `getCumulativeOfferData()` should be consistent with individual offer details.

- **Property 2:** Present Value Calculation
    - The sum of individual present values of offers should equal the total present value returned by `getPresentValue()` when called with `address(0)`.

- **Property 3:** Total Asset Value Consistency
    - The total asset value reported by the strategy should always be equal to the sum of the liquid balance, the present value of repoTokens, and the present value of pending offers.

The provided specifications of the mentioned functions were:

RepoTokenList

getCumulativeRepoTokenData

- **Purpose:** The function is designed to retrieve the cumulative sum of each `repoToken`'s remaining time to maturity (in seconds) weighted by `repoToken` balance AND the raw sum of each `repoToken`'s balance in the `Strategy` by iterating through the `repoTokenList`. This function returns three values:
    - `cumulativeWeightedTimeToMaturity`: cumulative sum of `repoTokens` held, weighted by remaining time to maturity in seconds
    - `cumulativeRepoTokenAmount`: cumulative sum of all `repoTokens` held
    - `found`: boolean on whether `repoToken` specified exists, if not, include for cases this method is called as a view function to simulate transactions
    - The optional `repoToken` and `repoTokenAmount` parameters are included to allow for simulating the addition of a new `repoToken` into the `Strategy`.

getPresentValue

- **Purpose:** Calculate the cumulative sum of the present value of each `repoToken` held by the `Strategy`. `Term` uses an `actual/360 day` count convention so that:

  ```
  Present value = notional * 1 / (1 + r * daysToMaturity / 360)
  ```

  - `totalPresentValue`: cumulative sum of the present value of each `repoToken` held by the portfolio

TermAuctionList

getCumulativeOfferData

- **Purpose:** The function is designed to retrieve the cumulative sum of each `offer`'s remaining time to maturity (in seconds) weighted by `offer.amount` AND the raw sum of each `offer`'s offer `amount` by iterating through the `pendingOffer` list. This function returns three values:
  - `cumulativeWeightedTimeToMaturity`: cumulative sum of outstanding offers weighted by remaining time to maturity in seconds
  - `cumulativeOfferAmount`: cumulative sum of outstanding offers
  - `found`: boolean on whether repoToken specified exists, if not, include for cases this method is called as a view function to simulate transactions
  - The optional `repoToken` and `repoTokenAmount` parameters are included to simulate the addition of a new `offer` or changes to an existing one.
  - Offers for completed auctions whose `repoToken` was not added to the `repoTokenList` yet because `the removeCompleted` function was not yet called should only be counted once per auction. The `offerAmount` for those offers should be calculated with `RepoTokenUtils.getNormalizedRepoTokenAmount`.
  - Offers for completed auctions whose `repoToken` is already in the repoTokenList should not be counted twice (once because the newly minted `repoTokens` will show up in the balance of the vault, and the second time because the pending offer has not been removed yet)

getPresentValue

- **Purpose:** Calculate the sum of all pending offers
    - `totalValue`: cumulative sum of all pending offers (except in the edge case or any other potential double-counting case)
    - Offers for completed auctions whose `repoToken` was not added to the `repoTokenList` yet because `the removeCompleted` function was not yet called should only be counted once per auction. The `offerAmount` for those offers should be calculated with `RepoTokenUtils.getNormalizedRepoTokenAmount`.
    - Offers for completed auctions whose `repoToken` is already in the repoTokenList should not be counted twice (once because the newly minted `repoTokens` will show up in the balance of the vault, and the second time because the pending offer has not been removed yet)

RepoTokenList

☑ **Test 1: The calculated cumulative `repoToken` data is consistent with the `repoTokens` in the list**

- **Objective:** This test verifies the cumulative weighted time to maturity and `repoToken` balances calculated by `getCumulativeRepoTokenData()` are consistent with the `repoTokens` stored in `repoTokenList`.
    - `cumulativeWeightedTimeToMaturity` is equal to the sum of each and every `repoToken` in the `Strategy` multiplied by its remaining time to maturity
    - `cumulativeRepoTokenAmount` value is equal to the sum of all repoToken balances in the Strategy

- **Current Status:** Verified and passed in **8m 25s** for bmc-depth 2.

- **Assumptions before function call:**
    - There are no duplicate tokens in the list (true because we proved this invariant before)
    - All tokens in the list have a positive balance (assumed for proof efficiency)

- **Details:**
    - Let `(cumTime, cumAmount) = getCumulativeRepoTokenData()`
    - Let `maturedTokensAmount = _filterMaturedTokensGetTotalValue(repoTokenList)` where `_filterMaturedTokensGetTotalValue` is a function that computes the

cumulative sum of `normalizedRepoTokenAmounts` of matured tokens and removes them from the list.
    - Let `(noMaturedTokensTime, noMaturedTokensAmount) = _cumulativeRepoTokenDataNotMatured` where `_cumulativeRepoTokenDataNotMatured` is a function that computes the cumulative `repoToken` data for non-matured tokens.

- **Assertions:**
    - The cumulative weighted time to maturity of the original list is equal to the cumulative weighted time to maturity after removing matured tokens: `cumTime == noMaturedTokensTime`

    - The cumulative `repoToken` amount of the original list is equal cumulative offer amount of matured `repoTokens` plus the cumulative offer amount of non-matured `repoTokens`:
      `cumAmount == maturedTokensAmount + noMaturedTokensAmount`

☑ **Test 2: The cumulative sum of `repoTokens` total value is consistent with the `repoTokens` in the list**

- **Objective:** This test verifies the value calculated by `getPresentValue` is equal to the sum of the present value of each and every `repoToken` in the `Strategy` multiplied by its remaining time to maturity.

- **Current Status:** Verified and passed in **11m 36s** for bmc-depth 2.

- **Assumptions before function call:**
    - There are no duplicate tokens in the list (true because we proved this invariant before)
    - `0 < purchaseTokenPrecision` - otherwise the function reverts due to division by 0

- **Details:**
    - Let `totalValue = getPresentValue()`
    - Let `maturedTokensTotalValue = _filterMaturedTokensGetTotalValue(repoTokenList)` where `_filterMaturedTokensGetTotalValue` is a function that computes the cumulative sum of `normalizedRepoTokenAmounts` of matured tokens and removes them from the list.
    - Let `noMaturedTokensTotalValue = _totalPresentValueNotMatured` where `_totalPresentValueNotMatured` is a function that computes the

cumulative `repoToken` present value for non-matured tokens.

- **Assertions:**
  - The cumulative present value of the original list is equal to the cumulative present value of matured `repoTokens` plus the cumulative present value of non-matured `repoTokens`:
    `totalValue == maturedTokensTotalValue + noMaturedTokensTotalValue`

TermAuctionList

☑ **Test 1: The calculated cumulative offer data is consistent with the `offers` in the list**

- **Objective:** This test verifies the cumulative weighted time to maturity and cumulative offer amount calculated by `getCumulativeOfferData()` are consistent with the `offers` stored in `termAuctionList`.
  - `cumulativeWeightedTimeToMaturity` value is equal to the sum of each `offer` multiplied by the remaining time to maturity (except in the edge case and any other potential double-counting case)
  - `cumulativeRepoTokenAmount` value is equal to the sum of all offers in `pendingOffers` (except in the edge case or any other potential double counting case).

- **Current Status:** Verified and passed in **29m 33s** for bmc-depth 2.

- **Assumptions before function call:**
  - There are no duplicate offers in the list (true because we proved this invariant before)
  - The locked offer amount for completed auctions is 0 (**if this is not true then the property does not hold**)
  - To prevent excessive branching on the proofs, which would cause a significant slowdown of proof executions, we assumed the following (notice the tests still pass without these assumptions but take longer time to execute):
    - The locked offer amount for non-completed auctions is greater than 0
    - There is no `offer` in the list whose `repoToken` has matured
    - The `repoTokens` for all offers have a positive `balance` and a positive `redemptionValue`

- **Details:**
  - Let `(cumulativeTime, cumulativeAmount) = getCumulativeOfferData()`
  - Let `(noCompletedAuctionsCumulativeTime, noCompletedAuctionsCumulativeAmount) = _filterCompletedAuctionsGetCumulativeOfferData` where `_filterCompletedAuctionsGetCumulativeOfferData` is a function that computes the cumulative weighted time to maturity and the cumulative offer amount for non-completed auctions and removes them from the list.
  - From the remaining list (with only completed auctions) remove the offers whose `repoToken` is already in the `repoTokenList`
  - From the remaining list remove offers for the same auction, i.e. there should be only one offer per auction
  - Let `(completedAuctionsCumulativeTime, completedAuctionsCumulativeAmount) = _getCumulativeOfferDataCompletedAuctions` where `_getCumulativeOfferDataCompletedAuctions` is a function that computes the cumulative weighted time to maturity and the cumulative `offer` amount for completed auctions

- **Assertions:**
  - The cumulative weighted time to maturity of the original list is equal to the cumulative weighted time to maturity of the non-completed auctions plus the cumulative weighted time to maturity of the list with only offers for completed (non-repeated) auctions whose repoToken is not on the repoTokenList:
    `cumulativeTime == noCompletedAuctionsCumulativeTime + completedAuctionsCumulativeTime`
  - The cumulative `offerAmount` of the original list is equal to the cumulative `offerAmount` of the non-completed auctions plus the cumulative `offerAmount` of the list with only offers for completed (non-repeated) auctions whose `repoToken` is not on the `repoTokenList`:
    `cumulativeAmount == noCompletedAuctionsCumulativeAmount + completedAuctionsCumulativeAmount`

☑ **Test 2: The sum of individual present values of offers should equal the total present value returned by `getPresentValue()` when called with `address(0)`**

- **Objective:** This test verifies the cumulative sum of the present value of offers calculated by `getPresentValue()` is consistent with the `offers` stored in `termAuctionList`.

- **Current Status:** Verified and passed in **31m 53s** for bmc-depth 2.

- **Assumptions before function call:**
  - There are no duplicate offers in the list (true because we proved this invariant before)
  - The locked offer amount for completed auctions is 0 (**if this is not true then the property does not hold**)
  - `0 < purchaseTokenPrecision` - otherwise the function reverts due to division by 0
  - The locked offer amount for non-completed auctions is greater than 0 (for proof efficiency, but the test still passes without this assumption

- **Details:**
  - Let `totalValue = getPresentValue()`
  - Let `noCompletedAuctionsTotalValue = _filterCompletedAuctionsGetTotalValue` where `_filterCompletedAuctionsGetTotalValue` is a function that computes the total present value for non-completed auctions and removes them from the list.
  - From the remaining list (with only completed auctions) remove the offers whose `repoToken` is already in the `repoTokenList`
  - From the remaining list remove offers for the same auction, i.e. there should be only one offer per auction
  - Let `completedAuctionsTotalValue = _getTotalValueCompletedAuctions` where `_getTotalValueCompletedAuctions` is a function that computes the total present value for completed auctions

- **Assertions:**
  - The total present value of the original list is equal to the total present value of the non-completed auctions plus the total present value of the list with only offers for completed (non-repeated) auctions whose repoToken is not on the repoTokenList:
    `totalValue == noCompletedAuctionsTotalValue + completedAuctionsTotalValue`

Findings During the Engagement

During the formal verification campaign, we identified the following issues:

- **Issue 1: Infinite loop**
  Impact: The function `insertSorted` may result in an infinite loop if there are two tokens with the same maturity and we try to insert the second one again.
  Resolution: The check for the maturity to insert should be:

  ```
  if (maturityToInsert < currentMaturity)
  ```

  Instead of:

  ```
  if (maturityToInsert <= currentMaturity)
  ```

  Fixed in commit: 7935718

- **Issue 2: `TermAuctionList.removeCompleted` function may become inoperable if `validateAndInsertRepoToken` reverts**
  Impact: This issue affects the functionality of the `TermAuctionList.removeCompleted` function. When **Resolution:** Make the functions `validateAndInsertRepoToken` and `validateRepoToken` return a `bool` whether the insertion was successful or not. The return value of these functions can be ignored in the `removeComplete`, but it should be changed in `Strategy.sol` to revert in the places where the result of calling these functions was false.
  Commit: a4385e8 and 9cb8e1c

- **Issue 3: `TermAuctionList.removeCompleted` function may become inoperable if `unlockOffers` reverts**
  Impact: This issue affects the functionality of the `TermAuctionList.removeCompleted` function. When iterating through the list of offers, if an auction is cancelled for withdrawal, then `TermAuctionOfferLocker.unlockOffers` function is called. This function can revert in some cases, meaning that the `removeCompleted` would revert forever, and it would become impossible to remove offers for completed or cancelled auctions from the list afterward.
  Resolution: Add a `try-catch` clause around the `unlockOffers` function call.
  Commit: 9cb8e1c

- **Issue 4: `TermAuctionList.insertSorted` does not preserve Invariant 7**
  Impact: Invariant 7 states that that there should be no offers in the list for completed or canceled auctions. However, there is no check that the new offer being inserted is not for a completed or canceled auction.

Resolution: Require that the auction is not completed or canceled for withdrawal.
Commit: c2f2d07

- **Issue 5: `TermAuctionList.removeCompleted` does not preserve Invariant 7 and 9**
  Impact: Invariant 7 states that there should be no offers in the list for completed or canceled auctions, and invariant 9 states that after the `removeCompleted` function is called all offers in the list for completed or canceled auctions should be removed from the list. However, if the list has more than 3 elements and there are 2 sequential elements whose auctions have been canceled or removed, the second offer to be removed will still be in the list after the function finishes.
  Resolution: When removing the `current` node from the list, the `current` node variable should be assigned to the `prev` node variable, so in the next iteration the `prev` node will be linked to the `next` node in the list.
  Commit: e533cd2 and 43df359

- **Issue 6: `TermAuctionList.removeCompleted` function may become inoperable if `getDiscountRate` reverts**
  Impact: This issue affects the functionality of the `TermAuctionList.removeCompleted` function. When iterating through the list of offers, if an auction is completed, the `removeCompleted` function will try to insert the `offer.repoToken` in the `repotokenListData` by calling the `validateAndInsertRepoToken`. This function calls `getDiscountRate` which can revert in some cases. This means that the `removeCompleted` would revert forever, and it would become impossible to remove offers for completed or canceled auctions from the list afterward.
  Resolution: Add a `try-catch` clause around the `getDiscountRate` function call.
  Commit: 7e44805 and e80505c

- **Issue 7: `repoTokenListData.nodes` and `repoTokenListData.discountRates` may become inconsistent**
  Impact: Some functions, such as `getPresentValue` and `getCumulativeOfferData` check whether a `repoToken` is on the list or not by checking that `repoTokenListData.discountRates[repoToken] == 0`. Such functions assume that `repoTokenListData.discountRates` and `repoTokenListData.discountRates` are consistent with each other, in the sense that if a `repoToken` is on the list then `repoTokenListData.discountRates[repoToken] != 0`. However, it can be the case that a `repoToken` is in the repo tokens list and `repoTokenListData.discountRates[repoToken] == 0`. This happens because the function validateAndInsertRepoToken, after inserting the repoToken in the list assigns

the `repoTokenListData.discountRates[repoToken]` to the value returned by `getDiscountRate(repoToken)`. However, there is no check that this value is different than 0.
The consequences are that the functions `getPresentValue` and `getCumulativeOfferData` will double count such `repoTokens` in some cases, **violating Properties 1 and 2**.
Resolution: Add an if statement and only insert the `repoToken` in the list if the value returned by `getDiscountRate` is different than 0.
Commit: 3919032

- **Issue 8: Some repoTokens may never be redeemed**
  Impact: The `repoveCompleted` function, iterates through the list of offers, and if the auction of some `offer` is completed and its repoToken has already matured, then the `repoToken` will not be inserted in the repoTokenListData, because the function `validateAndInsertRepoToken` returns false and does not insert the `repoToken` in the list. This `offer` is then removed from the `termAuctionListData`, which means that this `repoToken` was not redeemed and will never be, since it was not added to the `repoTokenListData`.
  Resolution: If the repoToken has already matured redeem it before removing the offer from the list.
  Commit: 3919032

Conclusion

This engagement is a good example of how formal verification can be crucial in ensuring the correctness of protocols. The specification of invariants and properties helped not only identify issues but also be confident that the desired properties hold under certain pre-conditions.

All the properties proven in the tests for the insert and removal operations of both list implementations and the properties for the get functions hold **under the assumptions** mentioned in each test. It is very important to ensure that when these functions are called the corresponding assumptions hold, otherwise preservation of the invariants is not guaranteed.

The most interesting issue in this engagement is Issue 5: **`TermAuctionList.removeCompleted` does not preserve Invariants 7 and 9**. First, it is not certain that this issue would be spotted with traditional code review, since it is a subtle bug that only manifests if we have lists with 3 elements or more and we try to remove 2 subsequent elements that are not the head of the list. In addition, and more importantly, the other formal verification tools in the market perform bounded model checking with bmc-depth 1 or at maximum 2. But, as already mentioned, this particular issue only manifests with lists with at least 3 elements.

An interesting follow-up engagement would be to prove these invariants for the functions where these functions are called. These functions are in `Strategy.sol`, where there are other interesting properties to prove. By following this approach, we would ensure the correctness of the entire system.