CERTORA

Formal Verification of OpenZeppelin (March - April 2022)

Summary

This document describes the specification and verification of OpenZeppelin’s contracts using the Certora Prover. The work was undertaken from March 2 to April 6, 2022. The latest commit that was reviewed and ran through the Certora Prover was `4088540a`.

The scope of this verification is OpenZeppelin’s governance system, particularly the following contracts:

- `ERC20Votes.sol`
- `ERC20FlashMint.sol`
- `ERC20Wrapper.sol`
- `TimelockController.sol`
- `draft-ERC721Votes.sol`
- `Votes.sol`
- `AccessControl.sol`
- `ERC1155.sol`

The Certora Prover proved the implementation of the contracts above is correct with respect to formal specifications written by the the Certora team. The team also performed a manual audit of these contracts.

The formal specifications focus on validating correct behavior of OpenZeppelin’s contracts as described by the OZ team and documentation. The rules verify valid states of a system, proper transitions between states, the solvency of the system and method specifications(unitTest-like rules).
The formal specifications have been submitted as a pull request against OpenZeppelin’s public git repository.

Main Issues Discovered

Severity: Medium

Issue:	**No check for `0` address [ERC1155]**
Description:	`from` address can be the `0` address in `safeTransferFrom` and `safeBatchTransferFrom`
Response:	Will be implemented in the next version.

Severity: Medium

Issue:	**Anyone can call `flashLoan` for a receiver [ERC20FlashMint]**
Description:	Anyone can call `flashLoan` for a `receiver` . An attacker can call `flashLoan` repeatedly on a receiver and drain its funds as the receiver contract has to pay back extra `fee` .
Response:	We've implemented EIP-3156. If a receiver pays a fee, they should validate the initiator in onFlashLoan

Severity: Informational

Issue:	**`Votes.sol` can only support token supply upto 2^224 - 1 [Checkpoints.sol push()]**
Description:	Since `Votes.sol` uses `Checkpoints.push` , which casts the new value to `uint224` , it is only able to support token supply up till `type(uint224).max` . If this is indeed the case, they should mention it in the comments as they have done it for `ERC20Votes.sol`
Response:	Votes is an abstraction of the mechanisme that was first introduced in `ERC20Votes` . Both are limited, by design, to uint224. We will improve `Votes` documentation to more clearly reflect that limitation.

Severity: Informational

Issue:	**Extra unnecessary require [Votes.sol getPastTotalSupply()]**
Description:	`require(blockNumber < block.number)` is checked twice when calling `getPastTotalSupply()`
Response:	The redundant require in `getPastTotalSupply` was indeed missed. The check should should indeed be removed from `Votes.sol` to save gas

Severity: Informational

Issue:	**Checkpoint Overflow [ERC20Votes.sol, draft-ERC721Votes.sol]**
Description:	Should the number of checkpoints go past 2^32 uint32 index used will no longer function properly resulting in a loss of votes. However, since the property that only one checkpoint per block number is held, this is not believed to be an issue in a realistic time frame
Response:	The “key” art of the Checkpoints is uint32 that is currently used to store block numbers. Having it overflow would be a real issue, but we consider it very unlikelly to ever overflow, at list considering the current chain design. Even if someone was to use block.timestamp based checkpoint to circumvent the unpredictable nature of block number on some L2s (which is a feature that our code doesn’t provide out of the box), that overflow would happen in the year 2106.

Severity: Low

Issue:	**Equal addresses of contract and `msg.sender` [ERC20Wrapper.sol depositFor()/withdrawTo()]**
Description:	Contract’s address(`address(this)`) can be equal to the `msg.sender` , thus, it’s posssible to deposit/withdraw without limits
Response:	The hability to mint `ERC20Wrapper` tokens without a counterpart, while apparently not serious, has the ability to create a serious inconsistency between the totalSupply and the amount of underlying token. This could confuse external observer. Additionnaly, extensions of the ERC20Wrapper might include functionnality that use these additionals “unbacked” tokens. We will add a check to prevent this.

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Summary of formal verification

Notations

✔ indicates the rule is formally verified on the latest reviewed commit.

✖
indicates the rule was violated under one of the tested versions of the code.

✍
indicates the rule is not yet formally specified.

⏱
indicates that some functions cannot be verified because the rules timed out
Footnotes describe any simplifications or assumptions used while verifying the
rules (beyond the general assumptions listed above).

In this document, verification conditions are either shown as logical formulas or Hoare triples of the form `{p} C {q}` . A verification condition given by a logical formula denotes an invariant that holds if every reachable state satisfies the condition.

Hoare triples of the form `{p} C {q}` hold if any non-reverting execution of program `C` that starts in a state satsifying the precondition `p` ends in a state satisfying the postcondition `q` . The notation `{p} C@withrevert {q}` is similar but applies to both reverting and non-reverting executions. Preconditions and postconditions are similar to the Solidity `require` and `assert` statements.

The syntax {p} (C1 ~ C2) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C1 and C2, and {q} describes the states after their executions. Notice that C1 and C2 result in different states.

Formulas relate the results of method calls. In most cases, these methods are getters defined in the contracts, but in some cases they are getters we have added to our harness or definitions provided in the rules file. Undefined variables in the formulas are treated as arbitrary: the rule is checked for every possible value of the variables.

Verification of ERC20FlashMint.sol

Summary

ERC20FlashMint is the extension of ERC20 to support flash loan operations.

Assumptions and simplifications for verification

- We unroll loops. Violations that require a loop to execute more than once will not be detected.

Properties

(✓) letsWatchItBurns

Check that if flashLoan() call is successful, then proper amount of tokens was burnt(fee + flashLoan amount).

```
{
    feeBefore = flashFee(token, amount);
}
    flashLoan(receiver, token, amount, data)
{
    amount + feeBefore == amount of burnt tokens
}
```

Verification of ERC20Wrapper.sol

Summary

ERC20Wrapper is the extension of ERC20 to support token wrapping. Users can exchange their “underlying tokens” to “wrapped tokens”.

Assumptions and simplifications for verification

- We unroll loops. Violations that require a loop to execute more than once will not be detected.

Properties

(✓) whatAboutTotal

The `totalSupply` of wrapped should be less than or equal to the `totalSupply` of underlying (assuming no external transfer).

```
totalSupply() ≤ underlyingTotalSupply()
```

(✓) underTotalAndContractBalanceOfCorrelation

The `totalSupply` of wrapper tokens is less than or equal to the underlying tokens held by the wrapper contract.

```
totalSupply() ≤ underlyingBalanceOf(erc20wrapper)
```

(✓) *depositForSpecBasic*

Check that values are updated correctly by `depositFor()` .

```
{
    msg.sender ≠ erc20wrapper
    ∧ underlyingContract ≠ erc20wrapper
    ∧ wrapperTotalBefore = totalSupply(e)
    ∧ underlyingTotalBefore = underlyingTotalSupply()
    ∧ underlyingThisBalanceBefore = underlyingBalanceOf(erc20wrapper)
}
    depositFor(account, amount)
{
    wrapperTotalAfter = totalSupply()
    ∧ underlyingTotalAfter = underlyingTotalSupply()
    ∧ underlyingThisBalanceAfter = underlyingBalanceOf(erc20wrapper)
    ∧ wrapperTotalBefore == wrapperTotalAfter - amount
    ∧ underlyingTotalBefore == underlyingTotalAfter
    ∧ underlyingThisBalanceBefore == underlyingThisBalanceAfter - amount
}
```

(✓) *depositForSpecWrapper*

Check that values are updated correctly by `depositFor()` .

```
{
    underlyingContract ≠ erc20wrapper
    ∧ wrapperUserBalanceBefore = balanceOf(account)
    ∧ wrapperSenderBalanceBefore = balanceOf(msg.sender)
}
    depositFor(account, amount)
{
    wrapperUserBalanceAfter = balanceOf(account)
    ∧ wrapperSenderBalanceAfter = balanceOf(msg.sender)
    ∧ (account == msg.sender
          ⇒ (wrapperUserBalanceBefore == wrapperSenderBalanceBefore
          ∧ wrapperUserBalanceAfter == wrapperSenderBalanceAfter
          ∧ wrapperUserBalanceBefore == wrapperUserBalanceAfter - amount)
    ∧ (account ≠ msg.sender
          ⇒ (wrapperUserBalanceBefore == wrapperUserBalanceAfter - amount
          ∧ wrapperSenderBalanceBefore == wrapperSenderBalanceAfter))
}
```

(✓) *depositForSpecUnderlying*

Check that values are updated correctly by `depositFor()` .

```
{
    msg.sender ≠ erc20wrapper
    ∧ underlyingContract ≠ erc20wrapper
    ∧ underlyingSenderBalanceBefore = underlyingBalanceOf(msg.sender)
    ∧ underlyingUserBalanceBefore = underlyingBalanceOf(account)
}
    depositFor(account, amount)
{
    underlyingSenderBalanceAfter = underlyingBalanceOf(msg.sender)
    ∧ underlyingUserBalanceAfter = underlyingBalanceOf(account)
    ∧ (account == msg.sender
          ⇒ (underlyingSenderBalanceBefore == underlyingUserBalanceBefore
          ∧ underlyingSenderBalanceAfter == underlyingUserBalanceAfter
          ∧ underlyingSenderBalanceBefore == underlyingSenderBalanceAfter
    ∧ (account ≠ msg.sender ∧ account == erc20wrapper
          ⇒ (underlyingSenderBalanceBefore == underlyingSenderBalanceAfte
          ∧ underlyingUserBalanceBefore == underlyingUserBalanceAfter - a
    ∧ (account ≠ msg.sender ∧ account ≠ erc20wrapper
          ⇒ (underlyingSenderBalanceBefore == underlyingSenderBalanceAfte
          ∧ underlyingUserBalanceBefore == underlyingUserBalanceAfter))
}
```

(✓) *withdrawToSpecBasic*

Check that values are updated correctly by `withdrawTo()` .

```
{
    underlyingContract ≠ erc20wrapper
    ∧ wrapperTotalBefore = totalSupply()
    ∧ underlyingTotalBefore = underlyingTotalSupply()
}
    withdrawTo(account, amount)
{
    wrapperTotalAfter = totalSupply()
    ∧ underlyingTotalAfter = underlyingTotalSupply()
    ∧ wrapperTotalBefore == wrapperTotalAfter + amount
    ∧ underlyingTotalBefore == underlyingTotalAfter
}
```

(✓) *withdrawToSpecWrapper*

Check that values are updated correctly by `withdrawTo()` .

```
{
    underlyingContract ≠ erc20wrapper
    ∧ wrapperUserBalanceBefore = balanceOf(account)
    ∧ wrapperSenderBalanceBefore = balanceOf(msg.sender)
}
    withdrawTo(account, amount)
{
    wrapperUserBalanceAfter = balanceOf(account)
    ∧ wrapperSenderBalanceAfter = balanceOf(msg.sender)
    ∧ (account == msg.sender
          ⇒ (wrapperUserBalanceBefore == wrapperSenderBalanceBefore
          ∧ wrapperUserBalanceAfter == wrapperSenderBalanceAfter
          ∧ wrapperUserBalanceBefore == wrapperUserBalanceAfter + amount)
    ∧ (account ≠ msg.sender
          ⇒ (wrapperSenderBalanceBefore == wrapperSenderBalanceAfter + am
          ∧ wrapperUserBalanceBefore == wrapperUserBalanceAfter))
}
```

(✓) depositForSpecUnderlying

Check that values are updated correctly by `depositFor()` .

```
{
    msg.sender ≠ erc20wrapper
    ∧ underlyingContract ≠ erc20wrapper
    ∧ underlyingSenderBalanceBefore = underlyingBalanceOf(msg.sender)
    ∧ underlyingUserBalanceBefore = underlyingBalanceOf(account)
    ∧ underlyingThisBalanceBefore = underlyingBalanceOf(erc20wrapper)
}
    withdrawTo(account, amount)
{
    underlyingSenderBalanceAfter = underlyingBalanceOf(msg.sender)
    ∧ underlyingUserBalanceAfter = underlyingBalanceOf(account)
    ∧ underlyingThisBalanceAfter = underlyingBalanceOf(erc20wrapper)
    ∧ (account == e.msg.sender
          ⇒ (underlyingSenderBalanceBefore == underlyingUserBalanceBefore
          ∧ underlyingSenderBalanceAfter == underlyingUserBalanceAfter
          ∧ underlyingUserBalanceBefore == underlyingUserBalanceAfter - a
    ∧ (account ≠ e.msg.sender ∧ account == erc20wrapper
          ⇒ (underlyingUserBalanceBefore == underlyingUserBalanceAfter
          ∧ underlyingSenderBalanceBefore == underlyingSenderBalanceAfter
    ∧ (account ≠ e.msg.sender ∧ account ≠ erc20wrapper
          ⇒ (underlyingUserBalanceBefore == underlyingUserBalanceAfter -
          ∧ underlyingSenderBalanceBefore == underlyingSenderBalanceAfter
          ∧ underlyingThisBalanceBefore == underlyingThisBalanceAfter + a
}
```

(✓) recoverSpec

Check that values are updated correctly by `_recover()` .

```
    {
        wrapperTotalBefore = totalSupply()
        ∧ wrapperUserBalanceBefore = balanceOf(account)
        ∧ wrapperSenderBalanceBefore = balanceOf(msg.sender)
    }
        _recover(account)
    {
        wrapperTotalAfter = totalSupply()
        ∧ wrapperUserBalanceAfter = balanceOf(account)
        ∧ wrapperSenderBalanceAfter = balanceOf(msg.sender)
        ∧ wrapperTotalBefore == wrapperTotalAfter - value
        ∧ (msg.sender == account
              ⇒ (wrapperUserBalanceBefore == wrapperSenderBalanceBefore
              ∧ wrapperUserBalanceAfter == wrapperSenderBalanceAfter
              ∧ wrapperUserBalanceBefore == wrapperUserBalanceAfter - value))
        ∧ (msg.sender ≠ account
              ⇒ (wrapperUserBalanceBefore == wrapperUserBalanceAfter - value
              ∧ wrapperSenderBalanceBefore == wrapperSenderBalanceAfter))
    }
```

Verification of TimelockController.sol

Summary

TimelockController is a contract module that is used to apply a timelock for operations, e.g. some time should pass before operation will be executed, thus users will have time to exit in case if they suspect something.

Assumptions and simplifications for verification

- We unroll loops. Violations that require a loop to execute more than three times will not be detected.

Properties

(✓) operationCheck

`isOperation()` correctness check.

```
getTimestamp(id) > 0 ⇔ isOperation(id)
```

(✓) pendingCheck

`isOperationPending()` correctness check.

```
getTimestamp(id) > _DONE_TIMESTAMP() ⇔ isOperationPending(id)
```

(✓) *readyCheck*

`isOperationReady()` correctness check.

```
(block.timestamp ≥ getTimestamp(id) ∧ getTimestamp(id) > 1)
        ⇔ isOperationReady(eid)
```

(✓) *doneCheck*

`isOperation()` correctness check.

```
getTimestamp(id) == _DONE_TIMESTAMP() ⇔ isOperationDone(id)
```

(✓) *unsetPendingTransitionGeneral*

Possible transitions: from unset to unset or pending only.

```
{
    ¬isOperation(id)
}
    < call to any function f >
{
    isOperationPending(id) ∨ ¬isOperation(id)
}
```

(✓) *unsetPendingTransitionMethods*

Possible transitions: from unset to pending via `schedule()` and `scheduleBatch()` only.

```
{
    ¬isOperation(id)
}
    < call to any function f >
{
    isOperationPending(id) ⇒ (f == schedule() ∨ f == scheduleBatch())
}
```

(✓) *readyDoneTransition*

Possible transitions: from ready to done via `execute()` and `executeBatch()` only.

```
{
    isOperationReady(id)
}
    < call to any function f >
{
    isOperationDone(id) ⇒ (f == execute() v f == executeBatch())
}
```

(✓) pendingCancelledTransition

Possible transitions: from pending to cancelled via `cancel()` only.

```
{
    isOperationPending(id)
}
    < call to any function f >
{
    ¬isOperation(id) ⇒ f == cancel()
}
```

(✓) doneToNothingTransition

Possible transitions: form done to done (once an operation is done, it remains done) only.

```
{
    isOperationDone(id)
}
    < call to any function f >
{
    isOperationDone(id)
}
```

(✓) minDelayOnlyChange

Only TimelockController contract can change `_minDelay()` .

```
{
    delayBefore = _minDelay()
}
    < call to any function f >
{
    delayAfter = _minDelay()
    ∧ delayBefore ≠ delayAfter
          ⇒ e.msg.sender == TimelockController
}
```

(✓) scheduleCheck

Scheduled operation’s timestamp == `block.timestamp + delay`.

```
{ }
    schedule(target, value, data, predecessor, salt, delay)
{
    getTimestamp(id) == block.timestamp + delay
}
```

(✓) cannotCallExecute

Cannot call `execute()` on a pending (not ready) operation.

```
{
    isOperationPending(id) ∧ ¬isOperationReady(id)
}
    execute@withrevert(target, value, data, predecessor, salt)
{
    lastReverted
}
```

(✓) executeRevertsFromUnset

Cannot call `execute()` on an unset operation.

```
{
    ¬isOperation(id)
}
    execute@withrevert(target, value, data, predecessor, salt)
{
    lastReverted
}
```

(✓) cancelledNotExecuted

Canceled operations cannot be executed → can’t move from canceled to done.

```
{
    ¬isOperation(id)
}
    < call to any function f >
{
    ¬isOperationDone(id)
}
```

(✓)[1] onlyProposer

Only proposer can schedule.

```
    {
        f == schedule() v scheduleBatch()
    }
        isCheckRoleReverted = _checkRole@withrevert(PROPOSER_ROLE());
        isScheduleReverted = < call to any function f >;
    {
        isCheckRoleReverted ⇒ isScheduleReverted
    }
```

(✓)[2] *cooldown*

If ready then has waited minimum period after was set to pending.

```
    {
        minDelay = getMinDelay()
    }
        schedule(e, target, value, data, predecessor, salt, delay);
        < call to any function f in environment e>
    {
        isOperationReady(e2, id) ⇒
                (e2.block.timestamp - e.block.timestamp ≥ minDelay)
    }
```

(✓) *scheduleChange*

`schedule()` should change only one id’s timestamp.

```
    {
        otherIdTimestampBefore = getTimestamp(otherId)
    }
        schedule(target, value, data, predecessor, salt, delay) for id
    {
        id ≠ otherId ⇒ otherIdTimestampBefore == getTimestamp(otherId)
    }
```

(✓) *executeChange*

`execute()` should change only one id’s timestamp.

```
    {
        otherIdTimestampBefore = getTimestamp(otherId)
    }
        execute(target, value, data, predecessor, salt) for id
    {
        id ≠ otherId ⇒ otherIdTimestampBefore == getTimestamp(otherId)
    }
```

(✓) *cancelChange*

`cancel()` should change only one id’s timestamp.

```
{
    otherIdTimestampBefore = getTimestamp(otherId)
}
    cancel(id)
{
    id ≠ otherId ⇒ otherIdTimestampBefore == getTimestamp(otherId)
}
```

Verification of ERC20Votes.sol

Summary

This contract is an extension for OpenZeppelin’s implementation of the ERC20 protocol. This extension handles the distribution of voting power based on a user’s owned tokens. This power may be delegated to others or to one’s own account. Notably this contract handles only the holding of votes, and not the use of them, which is left up to users of the contract.

Assumptions and simplifications for verification

- The DelegateBySig function was removed during verification due to a tool failure, to be fixed at a later date
- The MoveVotingPower function was altered to no longer use function pointers due to incompatability with the tool, the logic was left unchanged and it is assumed no bugs were introduced or removed with this change
- It is assumed for most rules that the number of checkpoints does not exceed one million, a very high number, but notably below the amount to overflow. Overflow of checkpoints can happen, however it is assumed the application will not run for the millenia needed for this to occur.

Definitions

`fromBlock` block.number stored for a given checkpoint

Properties

(✓) `votes_solvency`

Enough votes are in the system to supply votes for each user

```
totalSupply() > ∑getVotes(user)
```

- We do not directly call getVotes(user), but instead update an auxillary value whenever the votes are updated

(✓) blockNum_constrains_fromBlock

For any given `fromBlock` of a checkpoint, it is less than or equal to the block number of the operation reading/writing the checkpoint

```
ckptFromBlock(account, index) < e.block.number
```

- It is assumed `index` is within the bounds of checkpoints

(✓) fromBlock_constrains_numBlocks

The number of checkpoints for a current account is less than the latest `fromBlock`

```
numCheckpoints(account) ≤ ckptFromBlock(account, numCheckpoints(account
```

(✓) fromBlock_greaterThanEq_pos

For any given checkpoint of an account, its `fromBlock` is greater than its index in the array

```
ckptFromBlock(account, pos) ≥ pos
```

(✓) fromBlock_increasing

for any two given checkpoints of an account, the one with the larger index will also have the larger `fromBlock`

```
index1 > index2 ⇒ ckptFromBlock(account, index1) > ckptFromBlock(accoun
```

- The index must correspond to a valid checkpoint

(✗) maxInt_constrains_ckptsLength

The number of checkpoints for a given account may not overflow (maximum of 2^32)

```
unsafeNumCheckpoints(account) < 4294967295
```

- It is noted in the assumptions above that while this fails, the time it would take for the checkpoints to overflow is unrealistic. This is true if the condition that only one checkpoint per block is held for a given account, which is shown through the above invariants. The following rule ‘unique_checkpoints_rule’ shows this property in a different way.

(✓) unique_checkpoints_rule

If the last `fromBlock` recorded for an account does not change accross any function, neither can the number of checkpoints

```
{
     num_ckpts_ = numCheckpoints(account);
     fromBlock_ = num_ckpts_ == 0 ? 0 : ckptFromBlock(account, num_ckpt
}

<arbitrary function f>

{
    _num_ckpts = numCheckpoints(account);
    _fromBlock = _num_ckpts == 0 ? 0 : ckptFromBlock(account, _num_ckpt


    fromBlock_ == _fromBlock ⇒ num_ckpts_ == _num_ckpts v _num_ckpts ==
}
```

(✓) transfer_safe()

`transfer` may not alter the total number of votes and properly transfers the same amount of votes as token transfered from the sender’s delegate to the receiver’s

```
{
    delegator_pre = getVotes(delegates(delegator))
    delegatee_pre = getVotes(delegates(delegatee))
    totalVotes_pre = totalVotes()
}

    transferFrom(delegator, delegatee, amount)

{
    totalVotes_post = totalVotes()
    delegator_post = getVotes(delegates(delegator))
    delegatee_post = getVotes(delegates(delegatee))

    totalVotes_pre == totalVotes_post
    delegates(delegator) ≠ 0 ⇒ delegator_pre - delegator_post == amount
    delegates(delegatee) ≠ 0 ⇒ delegatee_post - delegatee_pre == amount
}
```

(✓) delegates_safe

functions other than the variations of `delegate()` may not change the stored delegate for a given account

```
{
    pre = delegates(account)
}

<arbitrary function f> // other than delegate()

{
   post = delegates(account)
   pre == post
}
```

(✓) delegatee_receives_votes

When delegating, the delegatee always receives the votes equal to the token balance of the delegator

```
 {
   delegator_bal = balanceOf(delegator)
   votes_= getVotes(delegatee)
}

   delegate(delegator, delegatee)

{
   _votes = getVotes(delegatee)

   _votes == votes_ + delegator_bal
}
```

- Assumes the delegator has not already delegated to the delegatee

(✓) previous_delegatee_votes_removed

When delegate is called, the account previously delegated to, denoted as `third` , loses votes equal to the token balance of the delegator. This may be one’s own account

```
    {
        delegator_bal = balanceOf(delegator)
        uint256 votes_ = getVotes(third)
    }

        delegate(delegator, delegatee);

    {
        _votes = getVotes(third)

        third ≠ 0x0 ⇒ _votes == votes_ - delegator_bal
     }
```

(✓) *delegate_contained*

Calling delegate will only affect the accounts of the delegator, delegatee, and (if applicable) the account of the previous delegatee.

```
    {
        votes_ = getVotes(other)
    }

        delegate(delegator, delegatee)

    {
        _votes = getVotes(other)

        votes_ == _votes
    }
```

- the arbitrary account other is set to be none of the delegator, delegatee, or delegatee before the function call

(✓) *delegate_no_frontrunning*

The above properties: `delegate_contained` , `previous_delegatee_votes_removed` , and `delegatee_receives_votes` still pass after an arbitrary function has been called

```
< arbitrary function f>

{
    delegator_bal = balanceOf(delegator)
    delegatee_votes_ = getVotes(delegatee)
    third_votes_ = getVotes(third)
    other_votes_ = getVotes(other)
}

delegate(delegator, delegatee)

{
    _delegatee_votes = getVotes(delegatee)
    _third_votes = getVotes(third)
    _other_votes = getVotes(other)

    _delegatee_votes == delegatee_votes_ + delegator_bal
    third ≠ 0 ⇒ _third_votes == third_votes_ - delegator_bal
    other_votes_ == _other_votes
}
```

(✓) mint_increases_totalSupply

Calling mint increases the total supply of token and the last total is saved properly in the `_totalSupplyCheckpoints`

```
{
    totalSupply_ = totalSupply()
}

    mint(account, amount)

{
    _totalSupply = totalSupply()

    _totalSupply == totalSupply_ + amount
    getPastTotalSupply(fromBlock) == totalSupply_
}
```

(✓) burn_decreases_totalSupply

Calling burn decreasees the total supply of token and properly saves the last total in the `_totalSupplyCheckpoints`

```
     {
         totalSupply_ = totalSupply()
     }

         burn(account, amount)

    {
        _totalSupply = totalSupply()

        _totalSupply == totalSupply_ - amount
    }
```

(✓) mint_doesnt_increase_totalVotes
Calling mint does not change the sum of all votes held

```
     {
         pre = ∑getVotes(user)
     }

         mint(account, ammount)

     {
         post = ∑getVotes(user)

         pre == post
     }
```

(✓) burn_doesnt_decrease_totalVotes
Calling burn does not change the sum of all votes held

```
    {
        pre = ∑getVotes(user)
    }

        burn (account, ammount)

    {
        post = ∑getVotes(user)

        pre == post
    }
```

Verification of draft-ERC721Votes.sol and Votes.sol

Summary

draft-ERC721Votes.sol is functionally quite analogous to erc20Votes and most rules and invariants were kept the same, however some rules and simplifications were adjusted based on the implimentation and differences of the ERC721 protocol.

Assumptions and simplifications for verification

Similar to ERC20 Votes, delegateBySig was removed, MovingDelegateVotes was split to remove function pointers, and number of checkpoints was capped to one million. Additionally, a mapping of users to their last checkpoint was added to assit with some rules, which uses the vote returned from calling push to Checkpoints.History, and the current block number. This change was due to the tool not being able to access information from within a nested struct in an external contract, and is assumed to be equivalent information.

Properties

(✓) `votes_solvency`

Enough votes are in the system to supply votes for each user

```
totalSupply() > ∑getVotes(user)
```

- Instead of calling `getVotes(user)` a hook is used, we assume this information to be equivalent

(✓) `blockNum_constrains_fromBlock`

For any given `fromBlock` of a checkpoint, it is less than the current block number. Essentially no future blocks can be set

```
ckptFromBlock(account, index) < e.block.number
```

- It is assumed `index` is within the bounds of checkpoints

(✓) `fromBlock_constrains_numBlocks`

The number of checkpoints for a current account is less than the latest `fromBlock`

```
numCheckpoints(account) ≤ ckptFromBlock(account, numCheckpoints(account
```

(✓) `fromBlock_greaterThanEq_pos`

For any given checkpoint of an account, its `fromBlock` is greater than its index in the array

```
ckptFromBlock(account, pos) ≥ pos
```

(✓) fromBlock_increasing

for any two given checkpoints of an account, the one with the larger index will also have the larger `fromBlock`

```
index1 > index2 ⇒ ckptFromBlock(account, index1) > ckptFromBlock(accoun
```

- The index must correspond to a valid checkpoint

(✗) maxInt_constrains_ckptsLength

The number of checkpoints for a given account may not overflow (maximum of 2^32)

```
unsafeNumCheckpoints(account) < 4294967295
```

- It is noted in the assumptions above that while this fails, the timeperiod it would take for the checkpoints to overflow is unrealistic. This is true if the condition that only one checkpoint per block is held for a given account, which is shown through the above invariants. The following rule also attempts to show this

(✓) unique_checkpoints_rule

If the last `fromBlock` recorded for an account does not change across any function, neither can the number of checkpoints

```
{
     num_ckpts_ = numCheckpoints(account);
     fromBlock_ = num_ckpts_ == 0 ? 0 : ckptFromBlock(account, num_ckpt
}

<arbitrary function f>

{
    _num_ckpts = numCheckpoints(account);
    _fromBlock = _num_ckpts == 0 ? 0 : ckptFromBlock(account, _num_ckpt


    fromBlock_ == _fromBlock ⇒ num_ckpts_ == _num_ckpts v _num_ckpts ==
}
```

(✓) transfer_safe()

`transfer` may not alter the total numer of votes and properly transfers the same amount of votes as token transfereed from the sender's delegate to the receiver's

```
    {
        delegator_pre = getVotes(delegates(delegator))
        delegatee_pre = getVotes(delegates(delegatee))
        totalVotes_pre = totalVotes()
    }

        transferFrom(delegator, delegatee, amount)

    {
        totalVotes_post = totalVotes()
        delegator_post = getVotes(delegates(delegator))
        delegatee_post = getVotes(deleegates(delegatee))

        totalVotes_pre == totalVotes_post
        delegates(a) ≠ 0 => votesA_pre - 1 == votesA_post
        delegates(b) ≠ 0 => votesB_pre + 1 == votesB_post
    }
```

(✓) delegates_safe

functions other than delegate may not change the stored delegate for a given account

```
     {
         pre = delegates(account)
     }

     <arbitrary function f>

     {
        post = delegates(account)
        pre == post
     }
```

(✓) delegatee_receives_votes

When delegating, the delegatee always receives the votes equal to the token balance of the delegator

```
 {
    delegator_bal = balanceOf(delegator)
    votes_= getVotes(delegatee)
}

    delegate(delegator, delegatee)

{
    _votes = getVotes(delegatee)

    _votes == votes_ + delegator_bal
}
```

- Assumes the delegator has not already delegated to the delegatee

(✓) *`previous_delegatee_votes_removed`*

The account previously delegated to, denoted as `third`, loses votes equal to the token balance of the delegator. This may be one’s own account

```
{
    delegator_bal = balanceOf(delegator)
    uint256 votes_ = getVotes(third)
}

    delegate(delegator, delegatee);

{
    _votes = getVotes(third)

    third ≠ 0x0 ⇒ _votes == votes_ - delegator_bal
 }
```

(✓) *`delegate_contained`*

Calling delegate will only affect the accounts of the delegator, delegatee, and (if applicable) the account of the previous delegatee.

```
    {
        votes_ = getVotes(other)
    }

        delegate(delegator, delegatee)

    {
        _votes = getVotes(other)

        votes_ == _votes
    }
```

- the arbitrary account other is set to be none of the delegator, delegatee, or delegatee before the function call

(✓) delegate_no_frontrunning
The above properties: `delegate_contained` , `previous_delegatee_votes_removed` , and `delegatee_receives_votes` still pass after an arbitrary function has been called

```
< arbitrary function f>

{
    delegator_bal = balanceOf(delegator)
    delegatee_votes_ = getVotes(delegatee)
    third_votes_ = getVotes(third)
    other_votes_ = getVotes(other)
}

delegate(delegator, delegatee)

{
    _delegatee_votes = getVotes(delegatee)
    _third_votes = getVotes(third)
    _other_votes = getVotes(other)

    _delegatee_votes == delegatee_votes_ + delegator_bal
    third ≠ 0 ⇒ _third_votes == third_votes_ - delegator_bal
    other_votes_ == _other_votes
```

Verification of AccessControl.sol

Summary

AccessControl is a contract module that is used to support access control mechanisms. Access control is based on defined roles (role-based), e.g. users with specific roles can only do specific operations.

Assumptions and simplifications for verification

- We unroll loops. Violations that require a loop to execute more than once will not be detected.

Properties

(✓)[1:1] `onlyRoleModifierCheckGrant`

If `onlyRole` modifier reverts then `grantRole()` reverts.

```
{
}
    checkRevert = _checkRole@withrevert(getRoleAdmin(role));
    grantRevert = grantRole@withrevert(role, account);
{
    checkRevert ⇒ grantRevert
}
```

(✓)[1:2] `onlyRoleModifierCheckRevoke`

If `onlyRole` modifier reverts then `revokeRole()` reverts.

```
{
}
    checkRevert = _checkRole@withrevert(getRoleAdmin(role));
    revokeRevert = revokeRole@withrevert(role, account);
{
    checkRevert ⇒ revokeRevert
}
```

(✓) `grantRoleEffect`

`grantRole()` does not affect any other account.

```
{
    account ≠ nonEffectedAcc
    ∧ hasRoleBefore = hasRole(anotherRole, nonEffectedAcc)
}
    grantRole(role, account)
{
    hasRoleAfter = hasRole(anotherRole, nonEffectedAcc)
    ∧ hasRoleBefore == hasRoleAfter
}
```

(✓) `revokeRoleEffect`

`revokeRole()` does not affect any other account.

```
{
    account ≠ nonEffectedAcc
    ∧ hasRoleBefore = hasRole(anotherRole, nonEffectedAcc)
}
    revokeRole(role, account)
{
    hasRoleAfter = hasRole(anotherRole, nonEffectedAcc)
    ∧ hasRoleBefore == hasRoleAfter
}
```

Verification of ERC1155.sol

Summary

ERC1155 is the token contract that can represent fungible and non-fungible token types. It's based on the EIP-1155.

Assumptions and simplifications for verification

- We unroll loops. Violations that require a loop to execute more than three times will not be detected.
- For batch version of functions we assume that arrays have length of 3 because we unroll loops three times, thus, we won't reach 4th elemnt of an array.
- `balanceOfBatch()` wasn't verified/used because of tool limitations.

Properties

Approval

(✓) `unexpectedAllowanceChange`
Any function, which is not `setApprovalForAll()` , should not change approval.

```
{
    approveBefore = isApprovedForAll(account, operator)
}
    < call to any function f except setApprovalForAll()>
{
    approveAfter = isApprovedForAll(account, operator)
    ∧ approveBefore == approveAfter
}
```

(✓) `onlyOwnerCanApprove`
Approval can be changed only by owner.

```
{
    aprovalBefore = isApprovedForAll(owner, operator)
}
    setApprovalForAll(operator, approved)
{
    aprovalAfter = isApprovedForAll(owner, operator)
    ∧ (aprovalBefore ≠ aprovalAfter ⇒ owner == msg.sender)
}
```

(✓) approvalRevertCases

Chech that `isApprovedForAll()` reverts in planned scenarios and no more (shouldn't revert at all).

```
{
}
    isApprovedForAll@withrevert(account, operator)
{
    ¬lastReverted
}
```

(✓) onlyOneAllowanceChange

`setApprovalForAll()` changes only one approval.

```
{
    userApproveBefore = isApprovedForAll(owner, user)
}
    setApprovalForAll(operator, approved)
{
    aprovalAfter = isApprovedForAll(owner, user)
    ∧ (userApproveBefore ≠ userApproveAfter
        ⇒ (e.msg.sender == owner ∧ operator == user))
}
```

Balance

(✓) unexpectedBalanceChange

Any function, which is not one of transfers, mints and burns, should not change balanceOf of a user.

```
{
    balanceBefore = balanceOf(from, id)
}
    < call to any function f except transfers, burns and mints>
{
    balanceAfter = balanceOf(from, id)
    ∧ balanceBefore == balanceAfter
}
```

(✓) balanceOfRevertCases

Check that `balanceOf()` reverts in planned scenarios (only if `account` is 0).

```
{
}
    balanceOf@withrevert(account, id)
{
    lastReverted ⇒ account == 0
}
```

(✓)[3] balanceOfBatchRevertCases

Check that `balanceOfBatch()` reverts in planned scenarios (only if at least one of `account` s is 0).

```
{
    accounts[0] == account1
    ∧ accounts[1] == account2
    ∧ accounts[2] == account3
}
    balanceOfBatch@withrevert(accounts, ids)
{
    lastReverted ⇒
        (account1 == 0 v account2 == 0 v account3 == 0)
}
```

Transfer

(✓) transferAdditivity

Additivity of `safeTransferFrom()` : `safeTransferFrom(a)` ; `safeTransferFrom(b)` has same effect as `safeTransferFrom(a+b)` .

```
amount == amount1 + amount2 ∧
safeTransferFrom(from, to, id, amount, data) ~ safeTransferFrom(from, t

Equivalent with respect to the balanceOf(from, id)
```

(✓)[4] *transferCorrectness*

`safeTransferFrom()` updates `from` and `to` balances correctly.

```
{
    to ≠ from
    ∧ fromBalanceBefore = balanceOf(from, id)
    ∧ toBalanceBefore = balanceOf(to, id)
}
    safeTransferFrom(from, to, id, amount, data)
{
    fromBalanceAfter = balanceOf(from, id)
    ∧ toBalanceAfter = balanceOf(to, id)
    ∧ fromBalanceBefore == fromBalanceAfter + amount
    ∧ toBalanceBefore == toBalanceAfter - amount
}
```

(✓)[4:1] *cannotTransferMoreSingle*

`safeTransferFrom()` cannot transfer more than `from` 's balance.

```
{
    balanceBefore = balanceOf(from, id)
}
    safeTransferFrom(from, to, id, amount, data)
{
    amount > balanceBefore ⇒ lastReverted
}
```

(✓)[4:2] *transferBalanceReduceEffect*

Sender calling `safeTransferFrom()` should only reduce `from` balance and not others' if sending amount is greater than 0.

```
{
    other ≠ to
    ∧ otherBalanceBefore = balanceOf(other, id)
}
    safeTransferFrom(from, to, id, amount, data)
{
    otherBalanceAfter = balanceOf(other, id)
    ∧ (from ≠ other
            ⇒ otherBalanceBefore == otherBalanceAfter)
}
```

(✓)[4:3] *transferBalanceIncreaseEffect*

Sender calling `safeTransferFrom()` should only reduce `to` balance and not others' if sending amount is greater than 0.

```
    {
        from ≠ other
        ∧ otherBalanceBefore = balanceOf(other, id)
    }
        safeTransferFrom(from, to, id, amount, data)
    {
        otherBalanceAfter = balanceOf(other, id)
        ∧ (other ≠ to
                ⇒ otherBalanceBefore == otherBalanceAfter)
    }
```

(✓)[4:4] *noTransferForNotApproved*

Cannot transfer without approval(`safeTransferFrom()` version).

```
    {
        from ≠ msg.sender
        ∧ approve = isApprovedForAll(from, msg.sender)
    }
        safeTransferFrom@withrevert(from, to, id, amount, data)
    {
        ¬approve ⇒ lastReverted
    }
```

(✓)[4:5] *noTransferEffectOnApproval*

`safeTransferFrom()` doesn't affect any approval.

```
    {
        approveBefore = isApprovedForAll(owner, operator)
    }
        safeTransferFrom(from, to, id, amount, data)
    {
        approveAfter = isApprovedForAll(owner, operator)
        ∧ approveBefore == approveAfter
    }
```

Mint

(✓) *mintAdditivity*

Additivity of `_mint()` : `_mint(a)` ; `_mint(b)` has same effect as `_mint(a+b)` .

```
    amount == amount1 + amount2 ∧
    _mint(to, id, amount, data) ~ _mint(to, id, amount1, data); _mint(to, i

    Equivalent with respect to the balanceOf(from, id)
```

(✓) *mintRevertCases*

Check that `_mint()` reverts in planned scenario(s) (only if `to` is 0).

```
{
}
    _mint@withrevert(to, id, amount, data)
{
    to == 0 ⇒ lastReverted
}
```

(✓) *mintBatchRevertCases*

Check that `_mintBatch()` reverts in planned scenario(s) (only if `to` is 0 or arrays have different length).

```
{
}
     _mintBatch@withrevert(to, ids, amounts, data)
{
    to == 0 v ids.length ≠ amounts.length ⇒ lastReverted
}
```

(✓)[4:6] *mintCorrectWork*

Check that `_mint()` updates `to` balance correctly.

```
{
    otherBalanceBefore = balanceOf(to, id)
}
    _mint(to, id, amount, data)
{
    otherBalanceAfter = balanceOf(to, id)
    ∧ otherBalanceBefore == otherBalanceAfter - amount
}
```

(✓)[4:7] *cantMintMoreSingle*

The user cannot `_mint()` more than max_uint256.

```
{
     balanceOf(to, id) + amount > max_uint256
}
     _mint@withrevert(to, id, amount, data)
{
     lastReverted
}
```

(✓)[4:8] *cantMintOtherBalances*

`_mint()` changes only `to` balance.

```
{
     otherBalanceBefore = balanceOf(other, id)
}
     _mint(to, id, amount, data)
{
     otherBalanceAfter = balanceOf(other, id)
     ∧ (other ≠ to ⇒ otherBalanceBefore == otherBalanceAfter)
}
```

Burn

(✓) *burnAdditivity*

Additivity of `_burn()` : `_burn(a)` ; `_burn(b)` has same effect as `_burn(a+b)` .

```
amount == amount1 + amount2 ∧
_burn(from, id, amount) ~ _burn(from, id, amount1); _burn(from, id, amo

Equivalent with respect to the balanceOf(from, id)
```

(✓) *burnRevertCases*

Chech that `_burn()` revertes in planned scenario(s) (only if `from` is 0).

```
{
}
    _burn@withrevert(rom, id, amount)
{
    from == 0 ⇒ lastReverted
}
```

(✓) *burnBatchRevertCases*

Chech that `_burn()` revertes in planned scenario(s) (only if `from` is 0 or arrays have different length).

```
{
}
    _burnBatch@withrevert(from, ids, amounts)
{
    (from == 0 v ids.length ≠ amounts.length) ⇒ lastReverted
}
```

(✓)[4:9] *burnCorrectWork*

Check that `_burn()` updates `from` balance correctly.

```
{
    otherBalanceBefore = balanceOf(from, id)
}
    _burn(from, id, amount)
{
    otherBalanceAfter = balanceOf(from, id)
    ∧ otherBalanceBefore == otherBalanceAfter + amount
}
```

(✓)[4:10] *cantBurnMoreSingle*

The user cannot `_burn()` more than they have.

```
{
     balanceOf(from, id) - amount < 0
}
     _burn@withrevert(from, id, amount)
{
     lastReverted
}
```

(✓)[4:11] *cantBurnOtherBalances*

`_burn()` changes only `from` balance.

```
{
     otherBalanceBefore = balanceOf(other, id)
}
     _burn(from, id, amount)
{
     otherBalanceAfter = balanceOf(other, id)
     ∧ (other ≠ from ⇒ otherBalanceBefore == otherBalanceAfter)
}
```

1. in CVL we can retreive whether method reverted or not and save it in bool variable. ↩ ↩ ↩

2. We use several environments to represent different env settings, e.g. block.timestamp for the same msg.sender, etc. More detailed: link ↩

3. There are only three accounts because we unroll loops three times, thus, there is no need to add more accounts. ↩

4. the same property was done for the batch version of the method. ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩ ↩