Formal Verification of Celo Core Contracts Release 4

Summary

This document describes the specification and verification of Celo using Certora Prover. The work was undertaken from May 13, 2021 - May 27, 2021. The rules are running continuously in Celo's CI system.

The scope of our verification was the updated Accounts contract and Governance contract.

The Certora Prover proved the implementation of the **Accounts and Governance** is correct with respect to the formal rules written by the **cLabs** and the Certora teams. During the verification process, the Certora Prover discovered minor bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high level specifications of **Accounts and Governance**. **All the rules are publicly available in** https://github.com/celo-org/celo-monorepo/blob/5cd7a58a4b41cdd00e07f7f007979f078d278802/packages/protocol/specs/accounts.spec **.**

Results for Accounts from Celo's CI

Results for Governance from Celo's CI

List of Main Issues Discovered

Severity: Medium

Issue:	Users' referendum votes may point to non-existing proposals. Certain functionalities could behave unexpectedly as a result.
Rules Broken:	Consistency of referendum votes
Description:	The `referendumVotes` map for each voter is keyed by an *index* to the array of dequeued proposals. If a proposal expires, the index may be changed to point to another proposal. This could lead to failures of `vote` and `revokeVotes`. In addition, the value returned by `getVoteRecord`, a getter that receives an index from `dequeued` and returns also the proposal ID, may not agree with the actual value in `dequeued`.
Mitigation:	If modifying `referendumVotes` structure is impossible, it may be better to improve error handling and logic of `revokeVotes` so that it does not fail in any event of inconsistency. In addition, functionalities that depend on `getVoteRecord` may be wrong, so these should be checked in client side, or add a require to guarantee the consistency of the returned value in `getVoteRecord`.

Severity: Low

Issue:	Getters for default signers may be misleading.
Rules Broken:	Getters for signers are in agreement.
Description:	For legacy roles, `getIndexedSigner` returns the account's signer for the legacy role, while `getDefaultSigner` returns the account itself.
Mitigation:	Make sure clients are aware of the distinction and not call `getDefaultSigner` for legacy roles. Alternatiely, require `getDefaultSigner` to run only on non-legacy roles.

Disclaimer

Notations

1. indicates the rule is formally verified on the latest commit. We write * when the rule was verified on a simplified version of the code (or under some assumptions).
2. ❌ indicates the rule was violated under one of the tested versions of the code.
3. indicates the rule is not yet formally specified.
4. indicates the rule is postponed (<due to other issue, low priority?>) .
5. We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.

The syntax {p} (C_1 C_2) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C_1 and C_2, and {q} describes the states after their executions. Notice that C_1 and C_2 result in different states. As a special case, C_1op C_2, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

Verification of Accounts contract

The Accounts contract was previously formally verified in the original launch of the Celo contracts, reported in May 2020: https://www.certora.com/pubs/CeloMay2020.pdf. For v5, the following proposed changes were implemented: https://github.com/celo-org/celo-proposals/blob/master/CIPs/cip-0010.md

Functions

isAccount(address a) : bool

Returns true if a is an account.

isSigner(address s) : bool

Returns true if *s* is a signer.

getAuthorizedBy(address s) : address

Returns the account that authorized the signer *s*, or 0 if none.

getDataEncryptionKeyLen(address a) : address

Returns the length of the data encryption key used by account *a*.

getNameLen(address a) : uint

Returns the length of the name of the account *a*.

getWalletAddress(address a) : address

Returns the wallet address of the account *a*.

getAttestationSigner(address a) : address

Returns the attestation signer of the account *a* if it has such a signer, otherwise returns *a*.

_getAttestationSigner(address a) : address

Returns the attestation signer of the account *a* if it has such a signer, otherwise returns 0.

getVoteSigner(address a) : address

Returns the vote signer of the account *a* if it has such a signer, otherwise returns *a*.

_getVoteSigner(address a) : address

Returns the vote signer of the account *a* if it has such a signer, otherwise returns 0.

getValidatorSigner(address a) : address

Returns the validator signer of the account *a* if it has such a signer, otherwise returns *a*.

_getValidatorSigner(address a) : address

Returns the validator signer of the account *a* if it has such a signer, otherwise returns 0.

getDefaultSigner(address a, bytes32 r) : address

Returns the default signer of the account *a* for a role *r* if it has such a signer, otherwise returns *a*.

_getDefaultSigner(address a, bytes32 r) : address

Returns the default signer of the account *a* for a role *r* if it has such a signer, otherwise returns 0.

isLegacyRole(bytes32 r) : bool

Returns true if *r* is the hash of a legacy role (Vote, Attestation or Validator).

isCompletedSignerAuthorization(address a, bytes32 r, address s) : bool

Returns true if the signer authorization process of signer *s* of an account *a* for a role *r* has been completed.

isStartedSignerAuthorization(address a, bytes32 r, address s) : bool

Returns true if the signer authorization process of signer *s* of an account *a* for a role *r* has been started.

getValidatorRole() : bytes32

Returns the hash representing the legacy Validator role.

getAttestationRole() : bytes32

Returns the hash representing the legacy Attestation role.

getVoteRole() : bytes32

Returns the hash representing the legacy Vote role.

authorizeValidatorSigner(address a, address s, Sig sig)

Authorizes *s* as the signer of *a* for the Validator role, using the signature *sig*.

authorizeVoteSigner(address a, address s, Sig sig)

Authorizes *s* as the signer of *a* for the Vote role, using the signature *sig*.

authorizeAttestationSigner(address a, address s, Sig sig)

Authorizes *s* as the signer of *a* for the Attestation role, using the signature *sig*.

authorizeSignerWithSig(address a, address s, bytes32 r, Sig sig)

Authorizes *s* as the signer of *a* for the role *r*, using the signature *sig*.

initialize(...): bool

Initialized the contract, returns true if successful.

createAccount(address a)

Creates an account for address *a*.

Properties

1. **If an account does not exist, it should not have a wallet address or a legacy signer**

```
¬isAccount(x)
  getWalletAddress(x) = 0
   _getAttestationSigner(x) = 0
   _getVoteSigner(x) = 0
   _getValidatorSigner(x) = 0
```

2. **An address cannot be both an account and a signer**

```
x  0  d  0  x  d  getAuthorizedBy(d) = x
    isAccount(x)  ¬isAccount(d)
```

3. **A signer of an account for the legacy roles should be authorized by the account**

```
x  0  d  0  x  d
   (_getAttestationSigner(x) = d
      v _getVoteSigner(x) = d
      v _getValidatorSigner(x) = d)
          isAccount(x)  getAuthroizedBy(d) = x
```

4. **A signer of an account for the new roles should be authorized by the account**

```
x  0  d  0  x  d  _getDefaultSigner(x, r) = d
  isAccount(x)  getAuthroizedBy(d) = x
```

5. **An address cannot be a signer of two addresses**

```
{ x  0  y  0  d  0  x  d  x  y  y  d
   isAccount(x)  getAuthorizedBy(d) = x
   isAccount(y)
   getIndexedSigner(x, r1) = d
   getIndexedSigner(y, r2)  d }
    op()
{ getIndexedSigner(y, r2)  d }
```

6. **An account may have more than one signer, for legacy roles**

```
{ x  0  d1  0  d2  0  x  d1  x  d2  d1  d2  isAccount(x) }
  ( authorizeValidatorSigner(x, d2, sig2) ~
    authorizeVoteSigner(x, d1, sig1); authorizeValidatorSigner(x,
d2, sig2) )
{ getAuthorizedBy(d1) = x  getAuthorizedBy(d2) = x }
```

7. **An account may have more than one signer, for new roles**

```
{ x  0  d1  0  d2  0    x  d1  x  d2  d1  d2  isAccount(x) }
  ( authorizeSignerWithSig(x, d2, r2, sig2) ~
      authorizeSignerWithSig(x, d1, r1, sig1);
      authorizeSignerWithSig(x, d2, r2, sig2)  )
{ getAuthorizedBy(d1) = x  getAuthorizedBy(d2) = x }
```

8. **The authorized-by relation cannot be removed**

```
{ account = getAuthorizedBy(signer) }
  op()
{ account = getAuthorizedBy(signer) v account = 0 }
```

9. **The Accounts contract can only be initialized once**

```
{ }
  success1 = initialize(args); success2 = initialize(args2)
{ success1  ¬sucesss2 }
```

10. **The only way to change the existence of an account is the createAccount function**

```
{ isAcc = isAccount(a) }
  op() where op  createAccount
{ isAcc = isAccount(a) }
```

11. **Legacy roles are not used when checking default signers**

    `isLegacyRole(role)  _getDefaultSigner(account, role) = 0`

12. **View functions conditions for no revert**
    All view functions are non payable. In addition, `hasAuthorizedSigner(address,string)` and `batchGetMetadataURL(address[])` were omitted because they can revert due to calldatasize mismatch. Full details

13. **Authorized by is anti-reflexive**
    `a  0  getAuthorizedBy(a)  a`

14. **When completing signer authorization, authorized-by relation must be set**
    `a  0  (isCompletedSignerAuthorization(a, r, s)  getAuthorizedBy(s) = a)`

15. **For signer authorizations, a signer can only appear as a signer of a single account**

```
a1  0  a2  0
  (isCompletedSignerAuthorization(a1, r1, s)
     isCompletedSignerAuthorization(a2, r2, s)
       a1 = a2)
```

16. **Only legacy roles can start and complete signer authorization in a single step**

```
{ ¬isStartedSignerAuthorization(a, r, s)
   ¬isCompletedSignerAuthorization(a, r, s) }
    op() where op  authorizeSignerWithSignature
{ isCompletedSignerAuthorization(a, r, s)  isLegacyRole(r) }
```

17. **Signer approval must be received for becoming a signer of a non-legacy role**

```
{ ¬isLegacyRole(r)  ¬isCompletedSignerAuthorization(a, r, s) }
    op() where op  authorizeSignerWithSignature
{ isCompletedSignerAuthorization(a, r, s)  caller of op = s }
```

18. **Starting an authorization is exclusive for the executing account**

```
{ ¬isStartedSignerAuthorization(a, r, s) }
    op()
{ isStartedSignerAuthorization(a, r, s)  caller of op = a }
```

19. **The only way authorized-by relation can be set is either using a signature, or by the signer being the caller**

```
{ getAuthorizedBy(s) = 0 }
  op(), where op  authorizeSignerWithSignature,
                  authorizeAttestationSigner,
                  authorizeValidatorSigner*,
                  authorizeVoteSigner
{ getAuthorizedBy(s) = 0 v caller of op = s }
```

20. **Can only change signer authorization under limited conditions: only the account can start a signer authorization for the account, only the account can remove an account's signer authorization, and only the signer or the account can complete their signer authorization**

```
{ started = isStartedSignerAuthorization(a, r, s)
   completed = isCompletedSignerAuthorization(a, r, s) }
    op()
{
  (started  isStartedSignerAuthorization(a, r, s)  caller of op =
a)
   (completed  ¬isCompletedSignerAuthorization(a, r, s)  caller of
op = a)
   ¬completed  isCompletedSignerAuthorization(a, r, s)  caller of
op = s)
}
```

21. **There is no functionality in the fallback function**

```
{}
  success = fallback()
{ ¬success }
```

22. **Getters for signers are in agreement** ❌

```
(r = getVoteRole()  getIndexedSigner(a, r) = getLegacySigner(a, r)
                    getIndexedSigner(a, r) = getVoteSigner(a))

(r = getValidatorRole()  getIndexedSigner(a, r) = getLegacySigner
(a, r)
                    getIndexedSigner(a, r) = getValidatorSigner
(a))

(r = getAttestationRole()  getIndexedSigner(a, r) = getLegacySigner
(a, r)
                    getIndexedSigner(a, r) = getAttestationSigner
(a))

(¬isLegacyRole(r)  getIndexedSigner(a, r) = getDefaultSigner(a, r))

(getIndexedSigner(a, r)  0
   getDefaultSigner(a, r)  0
   getLegacySigner(a, r)  0)

(¬isLegacyRole(r) // unexpected condition - rule is violated when
removed
    getIndexedSigner(a, r)  a  getDefaultSigner(a, r)  a)
```

23. **Account to signer and signer to account getters should agree**

```
a  s  (
(r = getVoteRole()
   s = getVoteSigner(a)  voteSignerToAccount(s) = a)

(r = getValidatorRole()
   s = getValidatorSigner(a)  validatorSignerToAccount(s) = a)

(r = getAttestationRole()
   s = getAttestationSigner(a)  attestationSignerToAccount(s) = a)

(getIndexedSigner(a, r) = s  signerToAccount(s) = a)
)
```

Table for View functions conditions for no revert

Function	Condition for no reverting
getLegacySigner(address a, bytes32 r)	isLeagcyRole(r)
hasLegacySigner(address a, bytes32 r)	isLeagcyRole(r)
signerToAccount(address a)	getAuthorizedBy(a) 0 v isAccount(a)
voteSignerToAccount(address a)	(getAuthorizedBy(a) 0 isSigner(getAuthorizedBy (a), a, getVoteRole())) v (getAuthorizedBy(a) = 0 isAccount(a)
validatorSignerToAccount(address a)	(getAuthorizedBy(a) 0 isSigner(getAuthorizedBy (a), a, getValidatorRole())) v (getAuthorizedBy(a) = 0 isAccount(a)
attestationSignerToAccount(address a)	(getAuthorizedBy(a) 0 isSigner(getAuthorizedBy (a), a, getAttestationRole())) v (getAuthorizedBy (a) = 0 isAccount(a)

Verification of Governance contract

Governance is the main managing contract of governance operations in Celo. The process of changing any parameter or code in the Celo suite of smart contracts is going through 4 steps: upvoting initial *proposal*s, approvals by the approval committee, referendum voting by owners of locked gold, and execution. Certora and cLabs have written rules describing the correctness of govrenance during its development in 2019. Following the changes made for v5, those rules were re-integrated into Celo's CI.

Functions

getProposal(uint256 p) : address, uint256, uint256, uint256

For the proposal with id *p*, return the proposer, deposit, timestamp and number of transactions that will be executed in execution phase.

stageDurations() : uint256, uint256, uint256

Returns the stage durations for proposals: time until approval stage, time until referendum stage, time until execution stage.

getUpvotes(uint256 p) : uint256

Returns the number of upvotes that a queued proposal *p* has received.

getUpvotedProposal(address a) : uint256

Returns the proposal upvoted by an account *a*.

getUpvoteRecord(address a) : uint256, uint256

Returns the proposal upvoted by an account *a* as well as the weight.

getQueueLength() : uint256

Returns the length of the queued proposals.

proposalCount() : uint256

Returns the total number of proposals created, used to generate the next proposal's ID.

getDequeuedLength() : uint256

Returns the length of the dequeued proposals.

getFromDequeued(uint256 i) : uint256

Returns the id of the proposal at index *i* of the dequeued array.

queueExpiry() : uint256

The time in seconds that a proposal can stay in the queue before expiring.

isApproved(uint256 p) : bool

Returns true if proposal *p* was approved.

isDequeuedProposalExpired(uint256 p) : bool

Returns true if the proposal *p* expired.

getVoteTotals(uint256 p) : uint256, uint256, uint256

Returns the yes, no and abstain votes for a proposal with id *p*.

approver() : address

Returns the address that is allowed to approve proposals in the approval stage.

getVoteRecord(address a, uint256 i) : uint256, uint256, uint256

Returns the vote record of account *a* in favor for index *i* in dequeued array.

Properties

1. **Consistency of dequeued array**

```
0  i < getDequeuedLength()  getFromDequeued(i)  proposalCount()
```

2. **Consistency of referendum votes** ❌

```
getVotedId(voter, indx)  0  proposalExists(getVotedId(voter, indx))
```

3. **Consistency of upvotes**

```
let proposalId, weight = getUpvotedRecord(a) in
  (proposalId  0  weight > 0) v (proposalId = 0  weight = 0)
```

4. **A user should not be able to upvote more than once**

```
{ accounts.voteSignerToAccount(u) = u
   upvotes = getUpvotes(p)
   upvotedProposal = getUpvotedProposal(u) }
    upvote(p) by caller u
{ upvotedProposal = p  getUpvotes(p)  upvotes }
```

5. **Proposal count is monotonic non-decreasing, and is increasing in** `propose`

```
{ count = proposalCount() }
  op()
{ proposalCount()  count  (op == propose  proposalCount() > count)
}
```

6. **Cannot vote in referendum stage before approval**

```
{ yes, no, abstain = getVoteTotals(p)  yes = 0  no = 0  abstain = 0
   approved = isApproved(p) }
    op()
{ yes', no', abstain' = getVoteTotals(p)
   (yes'  0 v no'  0 v abstain'  0)  approved }
```

7. **One cannot cancel votes (only change their type to yes/no/abstain)**

```
{ d = accounts.getVoteSigner(u)
   (recordValue, weight) = getVoteRecord(d, deqIndx)}
    success = vote(p, deqIndex, value) by caller u
{ value = NONE  ¬success  (recordValue, weight) = getvoteRecord(d,
deqIndx) }
```

8. **Votes can only be changed via** `vote` **and** `revokeVotes`

```
{ proposal, recordValue = getVoteRecord(u, deqIndx)
   (y,n,a) = getVoteTotals(proposal) }
  op() by caller u, where op  vote, revokeVotes
{ (y', n', a') = getVoteTotals(proposal)
   (recordValue  NONE  proposalExists(p)  y + n + a = y' + n' + a')
   (recordValue = YES  y' < y)
   (recordValue = NO  n' < n)
   (recordValue = ABSTAIN  a' < a)
   (recordValue = NONE  (¬proposalExists(p)  y' = 0  n' = 0  a' =
0))
        v (y' = y  n' = n  a' = a))
}
```

9. **Can only approve proposals that were promoted from the queue**

```
{ isDequeued = getFromDequeued(index) = p
   a = approver()
   ¬isApproved(p)
   ¬isDequeuedProposalExpired(p) }
  approve(p, index) by caller u
{ isApproved(p)  isDequeued v (¬isDequeued  getFromDequeued(index)
= p)
   isApproved(p)  u = a }
```

10. **An approved proposal cannot be disapproved, for as long as it not expired**

```
{ wasApproved = isApproved(p)  wasApproved  proposalExists(p) }
  op()
{ isApproved(p)  proposalExists(p)
   wasApproved  isApproved(p) v ¬proposalExists(p) }
```

11. **For a proposal to be executed, there must be more “yes” votes than “no” votes**

```
{ (y, n, _) = getVoteTotals(p) }
  success = execute(p, index)
{ n  y  ¬success }
```

12. **Only** `initialize`, `setApprovalStageDuration`, `setReferendumStageDuration` **and** `setExecutionStageDuration` **may change the stage durations**

```
{ (d1, d2, d3) = getStageDurations() }
  op(), where op  initialize, set*StageDuration
{ (d1, d2, d3) = getStageDurations() }
```

13. **Proposals can be added only via** `propose`

```
{ queueLen = getQueueLength() }
  op(), where op  propose
{ getQueueLength()  queueLen }
```

14. **The constitution can be changed only via** `setConstitution`

```
{ constitution = getConstitution(dest, sighash) }
  op(), where op  setConstitution
{ constitution = getConstitution(dest, sighash) }
```

15. `initialize` **can only be called once**

```
{ }
  success1 = initialize(args); success2 = initialize(args2)
{ success1  ¬sucesss2 }
```