Invariants

certora

Michael George

Stanford, August 2022

Invariants

What is an invariant?

- Something that doesn't change over time
- A property of the state (storage) that should be true between transactions
  - No side effects (view-only)

Examples:

Things that are invariants: properties of "valid" states

- The balance of the zero address is zero
- The total supply is the sum of all user balances
- Assets exceed liabilities (solvency)

Things that are not invariants: properties of transitions

- `transferFrom` reverts if the sender's allowance is 0
- A user's rewards can only increase

Invariants in CVL

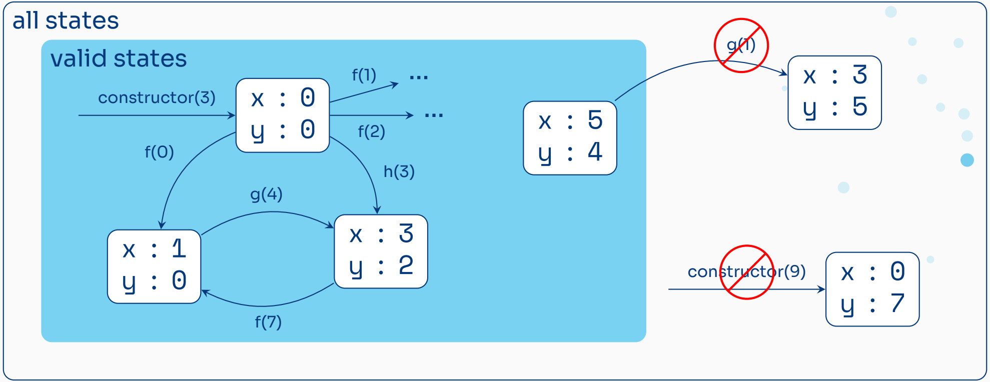
Writing an invariant in CVL:

```
/// The address 0x0 always has a balance of 0
invariant balanceOfZeroIsZero()
    balanceOf(0) == 0

/// The balance of a single user is always less than the total supply
invariant balanceBoundedBySupply(address a)
    balanceOf(a) <= totalSupply()
```

Checking invariants

- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid
- Need to check that transitions from valid states go to valid states

Verifying an invariant

```
/// The address 0x0 always has a balance of 0
invariant balanceOfZeroIsZero()
    balanceOf(0) == 0

{
    preserved with (env e) {
        require e.msg.sender != 0;
    }
}
```

(results link)

(results with preserved block)

- `preserved` blocks allow adding requirements to preservation checks
- WARNING: only use these for things that are always true!
  - ...examples of danger soon

BallGame Exercise (∼10 minutes)

`BallGame` is a simple implementation of keep away:

- Player 1 always passes to player 3
- Player 3 always passes to player 1
- Everyone else passes to player 2
- Ball starts with player 1
- Game is lost if player 2 gets the ball

Question: can player 2 ever get the ball?

- Exercise: Prove it!
- In `BallGame` directory:
  - Contract in `contracts/BallGame.sol`
  - Spec in `certora/specs/BallGame.spec`
  - Run using `sh certora/scripts/verifyBallGame.sh`

Solution walkthrough

Goal: player 2 never gets the ball

- First attempt:

```
invariant playerTwoNeverWins()
    ballPosition() != 2
```

Fails when `ballPosition` is 0! (results link)

- Second attempt: rule out bad case

```
invariant playerTwoNeverWins()
    ballPosition() != 2
{
    preserved with(env e) {
        require ballPosition() != 0;
    }
}
```

Fails with a different bad case! (results link)

certora

Third attempt: rule out more bad cases

```
invariant playerTwoNeverWins()
    ballPosition() != 2
{
    preserved with(env e) {
        require ballPosition() == 1 || ballPosition() == 3;
    }
}
```

Passes! (results link)
So the property holds …right?

```
//// contracts/BallGameBroken.sol

/// Move the ball to the next player,
/// based on who is currently holding it:
///   - player 1 will pass to player 3
///   - player 3 will pass to player 1
///   - everyone else will pass to player 2
///
/// @dev this version has a known bug
function pass() external {
    if (ballPosition == 1)
        ballPosition = 4;
    else if (ballPosition == 3)
        ballPosition = 1;
    else
        ballPosition = 2;
}
```

The rule still passes on the buggy code (results link)! Why?

- We ruled out the counterexample!
- We assumed something that we didn’t prove

certora

Fourth attempt: strengthening the invariant

- If ball position can only be 1 or 3, it can’t be 2; let’s prove that instead

```
invariant onlyGoodPlayers()
    ballPosition() == 1 || ballPosition() == 3
```

- Passes on our good code (results link)
  - No extra requirements, so property holds.
- Fails on our broken code (results link)
  - We catch the bad case

certora

Returning to original goal

- We wanted to prove `ballPosition() != 2`
- Instead we proved `ballPosition() == 1 || ballPosition() == 3`
- Seems stronger, but can we check?

```
/// The ball should never get to player 2
invariant playerTwoNeverWins()
    ballPosition() != 2
{
    preserved with (env e) {
        requireInvariant onlyGoodPlayers(); // was: require ballPosition() == 1 || ballPosition() == 3
    }
}
```

`requireInvariant` is shorthand for `require`

- `playerTwoNeverWins` still passes on correct code (link)
- Still passes on buggy version too (link)
- ...but it is much safer because we separately proved the requirement
- `requireInvariant` can be used anywhere `require` can, use it!

certora

Back to ERC20

Back to ERC20: Invariants about total supply

Let's prove invariants relating balances to total supply

- Individual user balances can't be larger than the total supply
- Total supply is the sum of user balances (next session)

Proving that each user balance is bounded by total supply

- First attempt (results link):

```
invariant balancesBoundedByTotalSupply(address a)
    balanceOf(a) <= totalSupply()
```

Fails on transfer:

  - although `a` starts with small balance, `b` doesn't necessarily!

- Second attempt: strengthen the invariant (results link)

```
invariant balancesBoundedByTotalSupply(address alice, address bob)
    balanceOf(alice) + balanceOf(bob) <= totalSupply()
```

Fails for the same reason!

  - `alice` and bob have small balances
  - but chuck might not!

- Fourth attempt: exercise (in 2 slides)
- Fifth (correct) attempt: next session

Summary

Things we covered in this session

- Invariants are properties of the state that don’t change over time
- Use `invariant` keyword to write invariants
  - Prover checks that constructor establishes invariant (instate)
  - Prover checks that methods maintain the invariant (preserve)
- `preserved` blocks are an “escape hatch” to tell the prover things you know
  - ...but this is dangerous!
  - Only require things that must be true
    - `requireInvariants`
    - platform assumptions (e.g. `msg.sender != 0`)
    - protocol assumptions (e.g. owner will never withdraw all the funds ...)
    - after writing specs, review your `preserved` blocks!
- Sometimes you need to strengthen invariants to prove them

Next session: strengthening bounded balance more and proving it

Exercise: Exploit the buggy rule

- Fourth attempt: use `preserved` blocks:

```
invariant balancesBoundedByTotalSupply(address alice, address bob)
    balanceOf(alice) + balanceOf(bob) <= totalSupply()
{
    preserved transfer(address recip, uint256 amount) with (env e) {
        require recip        == alice || recip        == bob;
        require e.msg.sender == alice || e.msg.sender == bob;
    }

    preserved transferFrom(address from, address to, uint256 amount) {
        require from == alice || from == bob;
        require to   == alice || to   == bob;
    }
}
```

  - Here `preserved` blocks apply to specific methods
- Rule passes (results link)
- Exercise: modify `ERC20.sol` to pass rule but violate invariant
- Note: I forgot to push this before we started!
  - In `ERC20Examples`:

```
git switch main
git pull
```

certora