Formal Verification with The Certora Prover

certora

Michael George

Product Director, Certora

DeFi Security Summit

Stanford, August 2022

Formal Verification with the Certora Prover

The Certora Prover is a tool for finding bugs in smart contracts

- Developers specify the intended behavior of the contract
  - Prover uses specs written in Certora Verification Language (CVL)
  - Specs are like unit tests

Formal Verification with the Certora Prover

The Certora Prover is a tool for finding bugs in smart contracts

- Developers specify the intended behavior of the contract
  - Prover uses specs written in Certora Verification Language (CVL)
  - Specs are like unit tests ...but infinitely more powerful

Formal Verification with the Certora Prover

The Certora Prover is a tool for finding bugs in smart contracts

- Developers specify the intended behavior of the contract
  - Prover uses specs written in Certora Verification Language (CVL)
  - Specs are like unit tests ...but infinitely more powerful
- The Prover checks that the contracts obey those properties

Formal Verification with the Certora Prover

The Certora Prover is a tool for finding bugs in smart contracts

- Developers specify the intended behavior of the contract
  - Prover uses specs written in Certora Verification Language (CVL)
  - Specs are like unit tests …but infinitely more powerful

- The Prover checks that the contracts obey those properties
  - in all circumstances! (every possible storage, every possible input)

Formal Verification with the Certora Prover

The Certora Prover is a tool for finding bugs in smart contracts

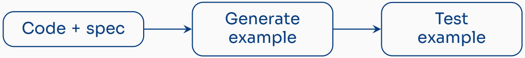
- Developers specify the intended behavior of the contract
  - Prover uses specs written in Certora Verification Language (CVL)
  - Specs are like unit tests …but infinitely more powerful
- The Prover checks that the contracts obey those properties
  - in all circumstances! (every possible storage, every possible input)

The Prover relies on results of decades of research in formal verification

- Both academic and industrial

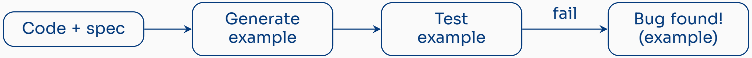
Formal Verification vs. Fuzzing

Fuzzing:



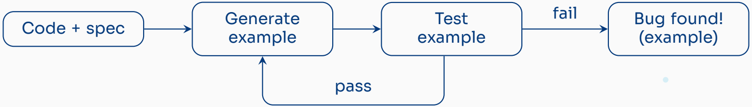
Formal Verification vs. Fuzzing

Fuzzing:



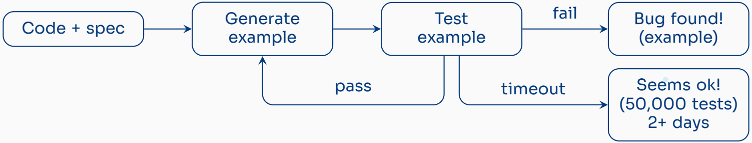
Formal Verification vs. Fuzzing

Fuzzing:



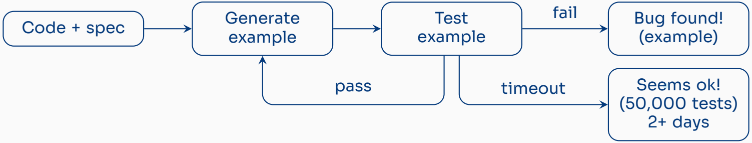
Formal Verification vs. Fuzzing

Fuzzing:



Formal Verification vs. Fuzzing

Fuzzing:



Code + spec

Generate example

Test example

fail

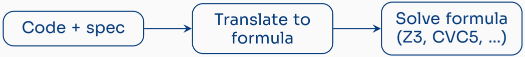
Bug found! (example)

pass

timeout

Seems ok! (50,000 tests) 2+ days

Formal verification (Certora Prover):



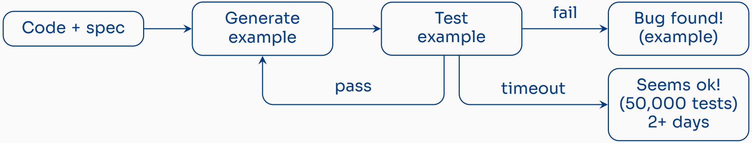
Code + spec

Translate to formula

Solve formula (Z3, CVC5, ...)

Formal Verification vs. Fuzzing

Fuzzing:



Code + spec

Generate example

Test example

fail

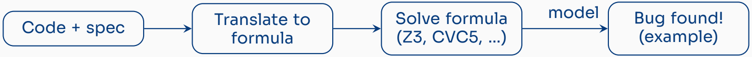
Bug found! (example)

pass

timeout

Seems ok! (50,000 tests) 2+ days

Formal verification (Certora Prover):



Code + spec

Translate to formula

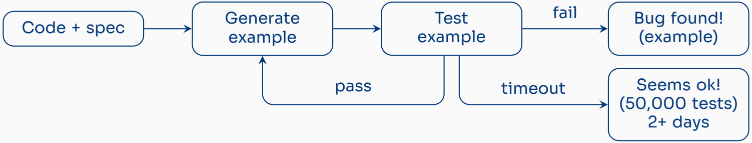
Solve formula (Z3, CVC5, ...)

model

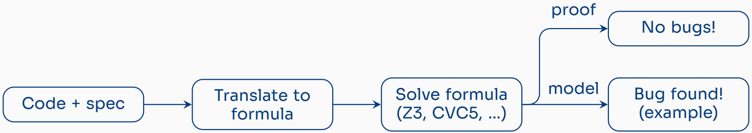
Bug found! (example)

Formal Verification vs. Fuzzing

Fuzzing:

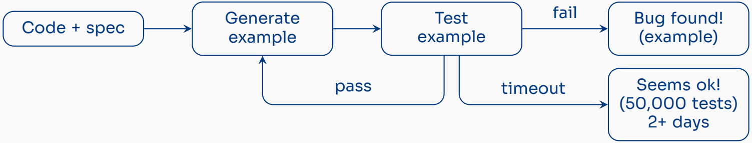


Formal verification (Certora Prover):



Formal Verification vs. Fuzzing

Fuzzing:



Code + spec

Generate example

Test example

fail

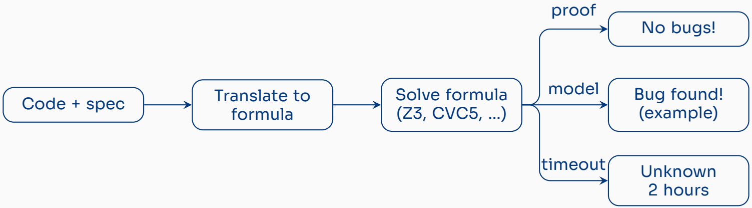
Bug found! (example)

pass

timeout

Seems ok! (50,000 tests) 2+ days

Formal verification (Certora Prover):



Code + spec

Translate to formula

Solve formula (Z3, CVC5, ...)

proof

No bugs!

model

Bug found! (example)

timeout

Unknown 2 hours

Certora Verficiation Language (CVL) Features:

- Solidity-like syntax
  - Specs can call methods, assign variables, define functions, ...

Certora Verficiation Language (CVL) Features:

- Solidity-like syntax
  - Specs can call methods, assign variables, define functions, ...
  - Use `require` to write preconditions, `assert` to describe behavior

Certora Verficiation Language (CVL) Features:

- Solidity-like syntax
  - Specs can call methods, assign variables, define functions, ...
  - Use `require` to write preconditions, `assert` to describe behavior
- Reason about arbitrary values
  - Prover considers every possible combination of values for undefined variables

Certora Verficiation Language (CVL) Features:

- Solidity-like syntax
  - Specs can call methods, assign variables, define functions, …
  - Use `require` to write preconditions, `assert` to describe behavior
- Reason about arbitrary values
  - Prover considers every possible combination of values for undefined variables
  - Prover considers every possible combination of values of storage variables

Certora Verficiation Language (CVL) Features:

- Solidity-like syntax
  - Specs can call methods, assign variables, define functions, …
  - Use `require` to write preconditions, `assert` to describe behavior
- Reason about arbitrary values
  - Prover considers every possible combination of values for undefined variables
  - Prover considers every possible combination of values of storage variables
- Syntax for calling an arbitrary method
  - e.g. if any method increases any user’s allowance, the user was the sender

Certora Verficiation Language (CVL) Features:

- Solidity-like syntax
  - Specs can call methods, assign variables, define functions, ...
  - Use `require` to write preconditions, `assert` to describe behavior
- Reason about arbitrary values
  - Prover considers every possible combination of values for undefined variables
  - Prover considers every possible combination of values of storage variables
- Syntax for calling an arbitrary method
  - e.g. if any method increases any user’s allowance, the user was the sender
- Access internal contract state
  - e.g. everytime `_balances[a]` changes, update `sum_balances`

Certora Verficiation Language (CVL) Features:

- Solidity-like syntax
  - Specs can call methods, assign variables, define functions, ...
  - Use `require` to write preconditions, `assert` to describe behavior
- Reason about arbitrary values
  - Prover considers every possible combination of values for undefined variables
  - Prover considers every possible combination of values of storage variables
- Syntax for calling an arbitrary method
  - e.g. if any method increases any user's allowance, the user was the sender
- Access internal contract state
  - e.g. everytime `_balances[a]` changes, update `sum_balances`
- Explicit syntax for writing state invariants

  -
    ```
    invariant totalSupplyBoundsBalances(address a)
        balanceOf(a) <= totalSupply()
    ```

Certora Verficiation Language (CVL) Features:

- Solidity-like syntax
  - Specs can call methods, assign variables, define functions, ...
  - Use `require` to write preconditions, `assert` to describe behavior
- Reason about arbitrary values
  - Prover considers every possible combination of values for undefined variables
  - Prover considers every possible combination of values of storage variables
- Syntax for calling an arbitrary method
  - e.g. if any method increases any user's allowance, the user was the sender
- Access internal contract state
  - e.g. everytime `_balances[a]` changes, update `sum_balances`
- Explicit syntax for writing state invariants

  -
    ```
    invariant totalSupplyBoundsBalances(address a)
        balanceOf(a) <= totalSupply()
    ```
- Reason about reverting paths
  - e.g. in emergency mode, `withdraw` never reverts unless ...

Certora Verficiation Language (CVL) Features:

- Solidity-like syntax
  - Specs can call methods, assign variables, define functions, ...
  - Use `require` to write preconditions, `assert` to describe behavior
- Reason about arbitrary values
  - Prover considers every possible combination of values for undefined variables
  - Prover considers every possible combination of values of storage variables
- Syntax for calling an arbitrary method
  - e.g. if any method increases any user’s allowance, the user was the sender
- Access internal contract state
  - e.g. everytime `_balances[a]` changes, update `sum_balances`
- Explicit syntax for writing state invariants

  -
    ```
    invariant totalSupplyBoundsBalances(address a)
        balanceOf(a) <= totalSupply()
    ```
- Reason about reverting paths
  - e.g. in emergency mode, `withdraw` never reverts unless ...
- Rewind storage to a previous state
  - e.g. Rerunning with more permissions doesn’t cause revert

certora

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {








    transfer(   recip, amount);






    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

certora

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    transfer(   recip, amount);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    mathint balance_sender_before = balanceOf(sender);


    transfer(   recip, amount);





    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

certora

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(   recip, amount);




    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(   recip, amount);

    mathint balance_sender_after = balanceOf(sender);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(   recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;


    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(   recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

certora

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;


    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Example / demo

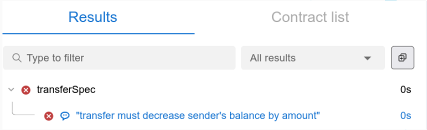
```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

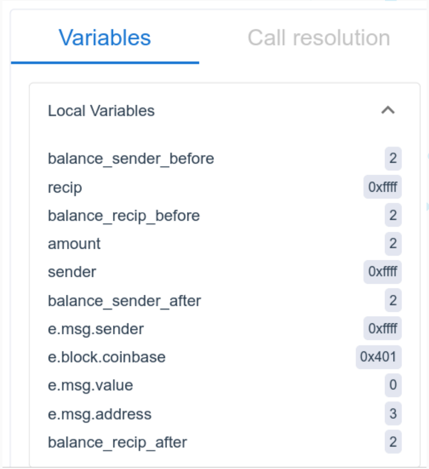


transferSpec 0s

"transfer must decrease sender's balance by amount" 0s

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```



Local Variables

balance_sender_before	2
recip	0xffff
balance_recip_before	2
amount	2
sender	0xffff
balance_sender_after	2
e.msg.sender	0xffff
e.block.coinbase	0x401
e.msg.value	0
e.msg.address	3
balance_recip_after	2

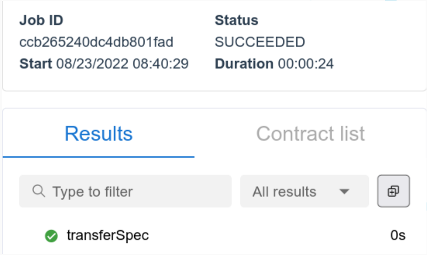
Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);

    require sender != recip;

    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

certora

Example / demo

```
/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);

    require sender != recip;

    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```



Job ID ccb265240dc4db801fad

Status SUCCEEDED

Start 08/23/2022 08:40:29

Duration 00:00:24

✓ transferSpec 0s

Verification is only as good as the specification

We’re currently developing tools to find bugs in specifications:

Verification is only as good as the specification

We're currently developing tools to find bugs in specifications:

- Vacuity checking, tautology checking
  - Sanity checks to flag rules that couldn't possibly catch bugs

Verification is only as good as the specification

We’re currently developing tools to find bugs in specifications:

- Vacuity checking, tautology checking
  - Sanity checks to flag rules that couldn’t possibly catch bugs
- Bug injection
  - Verifying versions with known bugs against the spec
  - Specs with good coverage should catch them!

Verification is only as good as the specification

We're currently developing tools to find bugs in specifications:

- Vacuity checking, tautology checking
  - Sanity checks to flag rules that couldn't possibly catch bugs
- Bug injection
  - Verifying versions with known bugs against the spec
  - Specs with good coverage should catch them!
- Mututation testing
  - Automatically change the code in ways that probably introduce bugs
  - e.g. remove method decorators, change `require` statements, ...
  - Specs with good coverage should catch them!

certora

Thank you!

Questions?

`https://demo.certora.com`

certora

Setting up the Certora Prover

Michael George

Stanford, August 2022

Workshop overview

Today: using the Certora Prover

- Installation
- Basic rules
- Lunch
- Invariants
- Ghosts

Tomorrow: Background and practical use

- How the Prover works
- Introduction to AAVE Governance token
- Lunch
- Systematic specification design
- Work session
- Closing

Logistics

For synchronous watchers (in person / streaming):

- Ask questions! In person or on our Discord in `#stanford-certora-workshop`
- Slides are dense; we'll post on discord
- Follow along; finished examples are in the repository
- We'll do lots of exercises
- Recordings will be available on Certora youtube channel

For asynchronous watchers:

- Ask questions! On the forum: `https://forum.certora.com/`
- Slides and repository are linked in the comments

Installing the Prover and Examples

1. Clone and update the Examples repo
   - `https://github.com/Certora/Examples`
2. Update the submodules
   - `git submodule update --init`
3. Install the Certora Prover

   Option 1: VSCode + Docker

   3.1 Install VSCode

   3.2 Install Docker Desktop

   3.3 Install “Remote - Containers” VSCode extension

   3.4 Open the ERC20Example folder in VSCode

   3.5 View → Command Palette → Reopen In Container

   Option 2: Local install

   3.1 Install Python

   3.2 Install Java JRE

   3.3 Install solc-select

   3.4 `pip install certora-cli`
4. Set your `CERTORAKEY` to the key we sent you
   - in a terminal, run “`export CERTORAKEY=<key we sent you>`”
5. Verify the ERC20 example
   - in a terminal, change to `ERC20Examples` directory
   - `sh certora/scripts/verifyERC20.spec`
   - view the report link that is printed

CVL: Basic Rules

certora

Michael George

Stanford, August 2022

Overview for this session

Basic rules for ERC20 contracts

- Presentation: writing and debugging rules
  - `transfer` changes balances appropriately
  - `transfer` reverts when it should
  - `transfer` doesn't revert unexpectedly
- Exercise: similar rules for `transferFrom`

Generalized (parametric) rules

- Presentation: rules that apply to all methods
  - Only the owner can increase their `allowance`
  - The owner only changes their allowance deliberately
- Exercise: similar rules for `balanceOf`

ERC20 transfer and balanceOf

The first properties we’d like to test are described in the interface:

```
//// contracts/IERC20.spec

/**
 * Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20 {


    /**
     * Moves `amount` tokens from the caller's account to `recipient`.
     */
    function transfer(address recipient, uint256 amount)
        external
        returns(bool);


    /**
     * Returns the amount of tokens owned by `account`.
     */
    function balanceOf(address account)
        external view
        returns(uint256);


    ...
}
```

certora

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec




/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {








    transfer(   recip, amount);







    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

certora

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec



/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    transfer(   recip, amount);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec



/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    mathint balance_sender_before = balanceOf(sender);


    transfer(   recip, amount);




    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec



/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(   recip, amount);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

certora

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec




/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(   recip, amount);

    mathint balance_sender_after = balanceOf(sender);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

certora

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec



/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;



    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(   recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec




/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;


    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(   recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

certora

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec




/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;


    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec



/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

certora

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec

methods {
    balanceOf(address) returns (uint) envfree
}

/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec

methods {
    balanceOf(address) returns (uint) envfree
}

/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);



    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

(results link)

Specifying transfer in CVL (unit-test-style rules)

(results link)

```
//// certora/specs/ERC20.spec

methods {
    balanceOf(address) returns (uint) envfree
}

/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);

    require sender != recip;

    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

Specifying transfer in CVL (unit-test-style rules)

```
//// certora/specs/ERC20.spec

methods {
    balanceOf(address) returns (uint) envfree
}

/// Transfer must move `amount` tokens from
/// the caller's account to `recipient`.
rule transferSpec {
    address sender; address recip; uint amount;

    env e;
    require e.msg.sender == sender;

    mathint balance_sender_before = balanceOf(sender);
    mathint balance_recip_before  = balanceOf(recip);

    transfer(e, recip, amount);

    mathint balance_sender_after = balanceOf(sender);
    mathint balance_recip_after  = balanceOf(recip);

    require sender != recip;

    assert balance_sender_after == balance_sender_before - amount,
        "transfer must decrease sender's balance by amount";

    assert balance_recip_after  == balance_recip_before  + amount,
        "transfer must increase recipient's balance by amount";
}
```

(results link)

(second link)

What about revert?

So far:

- Transfer reduces sender’s balance by `amount`
- Transfer increases recipient’s balance by `amount`

What about revert?

So far:

- Transfer reduces sender’s balance by `amount`
- Transfer increases recipient’s balance by `amount`

What if sender’s balance is less than `amount`?

What about revert?

So far:

- Transfer reduces sender’s balance by `amount`
- Transfer increases recipient’s balance by `amount`

What if sender’s balance is less than `amount`?

- Transaction reverts
- No balances change!
- Why doesn’t this violate the rule?

What about revert?

So far:

- Transfer reduces sender’s balance by `amount`
- Transfer increases recipient’s balance by `amount`

What if sender’s balance is less than `amount`?

- Transaction reverts
- No balances change!
- Why doesn’t this violate the rule?

Answer: by default, Prover ignores reverting paths.

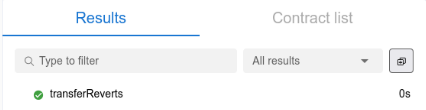
- we can override this behavior to reason about reverting

transfer revert conditions

```
//// certora/specs/ERC20.spec

/// Transfer must revert if the sender's balance is too small
rule transferReverts {
    env e; address recip; uint amount;

    require balanceOf(e.msg.sender) < amount;

    transfer@withrevert(e, recip, amount);

    assert lastReverted,
        "transfer(recip,amount) must revert if sender's balance is less than `amount`";
}
```

transfer revert conditions

```
//// certora/specs/ERC20.spec

/// Transfer must revert if the sender's balance is too small
rule transferReverts {
    env e; address recip; uint amount;

    require balanceOf(e.msg.sender) < amount;

    transfer@withrevert(e, recip, amount);

    assert lastReverted,
        "transfer(recip,amount) must revert if sender's balance is less than `amount`";
}
```

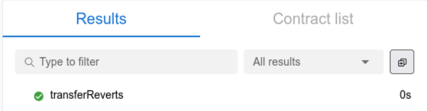


✓ transferReverts 0s

(results link)

transfer revert conditions

```
//// certora/specs/ERC20.spec

/// Transfer must revert if the sender's balance is too small
rule transferReverts {
    env e; address recip; uint amount;

    require balanceOf(e.msg.sender) < amount;

    transfer@withrevert(e, recip, amount);

    assert lastReverted,
        "transfer(recip,amount) must revert if sender's balance is less than `amount`";
}
```



transferReverts 0s

(results link)

Reasoning about reverts:

- Use `f@withrevert(...)` to consider paths where `f` reverts
- Use `lastReverted` to determine whether last call reverted
  - Warning: it is always the last call!
  - save it if you need to make another call

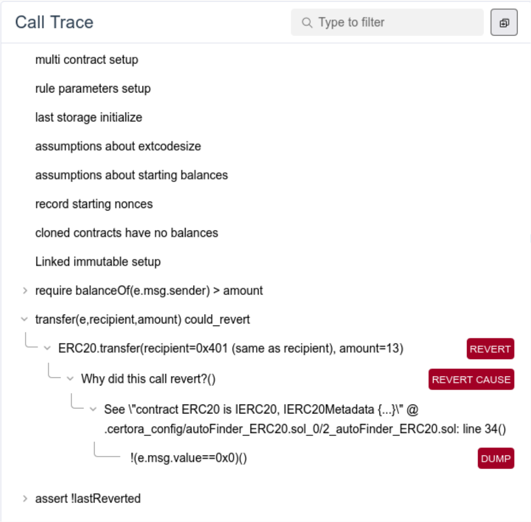
Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///
///
///
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;





    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```

certora

Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///     - the sender doesn't have enough funds
///
///
///
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;



    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```



Call Trace

```
multi contract setup
rule parameters setup
last storage initialize
assumptions about extcodesize
assumptions about starting balances
record starting nonces
cloned contracts have no balances
Linked immutable setup
› require balanceOf(e.msg.sender) > amount
⌄ transfer(e,recipient,amount) could_revert
  └ ⌄ ERC20.transfer(recipient=0x401 (same as recipient), amount=13)        REVERT
      └ ⌄ Why did this call revert?()                                       REVERT CAUSE
          └ ⌄ See \"contract ERC20 is IERC20, IERC20Metadata {...}\" @
              .certora_config/autoFinder_ERC20.sol_0/2_autoFinder_ERC20.sol: line 34()
              └── !(e.msg.value==0x0)()                                     DUMP
› assert !lastReverted
```

Proving transfer doesn’t revert (liveness rules)

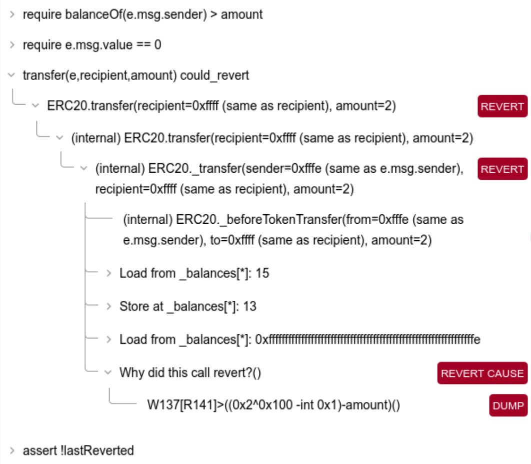
```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///
///
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;





    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```

Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///
///
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;




    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```

Proving transfer doesn't revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///
///
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;



    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```



```
> require balanceOf(e.msg.sender) > amount
> require e.msg.value == 0
v transfer(e,recipient,amount) could_revert
  v ERC20.transfer(recipient=0xffff (same as recipient), amount=2)                          REVERT
    v (internal) ERC20.transfer(recipient=0xffff (same as recipient), amount=2)
      v (internal) ERC20._transfer(sender=0xfffe (same as e.msg.sender),                    REVERT
        recipient=0xffff (same as recipient), amount=2)
          (internal) ERC20._beforeTokenTransfer(from=0xfffe (same as
          e.msg.sender), to=0xffff (same as recipient), amount=2)
        > Load from _balances[*]: 15
        > Store at _balances[*]: 13
        > Load from _balances[*]: 0xfffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe
        v Why did this call revert?()                                                 REVERT CAUSE
            W137[R141]>((0x2^0x100 -int 0x1)-amount)()                                      DUMP
> assert !lastReverted
```

Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///    - or the recipient's balance would overflow,
///
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;




    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```

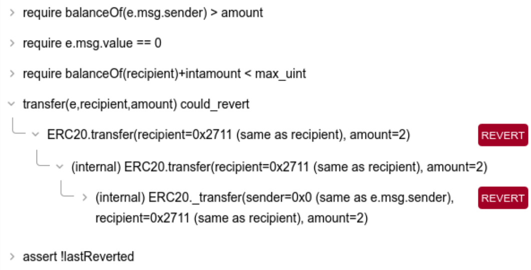
certora

Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///    - or the recipient's balance would overflow,
///
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;
    require balanceOf(recipient) + amount < max_uint;



    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```

Proving transfer doesn't revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///    - or the recipient's balance would overflow,
///
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;
    require balanceOf(recipient) + amount < max_uint;



    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```



```
> require balanceOf(e.msg.sender) > amount
> require e.msg.value == 0
> require balanceOf(recipient)+intamount < max_uint
v transfer(e,recipient,amount) could_revert
  └ v ERC20.transfer(recipient=0x2711 (same as recipient), amount=2)        REVERT
      └ v (internal) ERC20.transfer(recipient=0x2711 (same as recipient), amount=2)
          └ > (internal) ERC20._transfer(sender=0x0 (same as e.msg.sender),   REVERT
              recipient=0x2711 (same as recipient), amount=2)
> assert !lastReverted
```

Proving transfer doesn't revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///    - or the recipient's balance would overflow,
///    - or the message sender is 0
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;
    require balanceOf(recipient) + amount < max_uint;



    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```

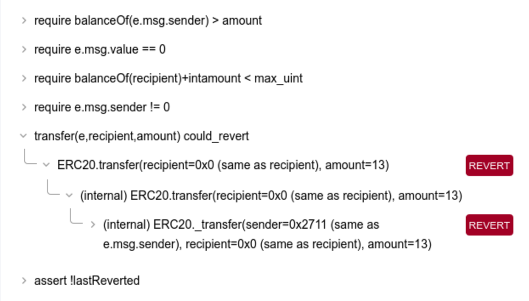
certora

Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///    - or the recipient's balance would overflow,
///    - or the message sender is 0
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;
    require balanceOf(recipient) + amount < max_uint;
    require e.msg.sender != 0;


    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```

Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///    - or the recipient's balance would overflow,
///    - or the message sender is 0
///
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;
    require balanceOf(recipient) + amount < max_uint;
    require e.msg.sender != 0;


    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```



```
> require balanceOf(e.msg.sender) > amount
> require e.msg.value == 0
> require balanceOf(recipient)+intamount < max_uint
> require e.msg.sender != 0
v transfer(e,recipient,amount) could_revert
  └ v ERC20.transfer(recipient=0x0 (same as recipient), amount=13)                REVERT
      └ v (internal) ERC20.transfer(recipient=0x0 (same as recipient), amount=13)
          └ > (internal) ERC20._transfer(sender=0x2711 (same as                   REVERT
              e.msg.sender), recipient=0x0 (same as recipient), amount=13)
> assert !lastReverted
```

Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///    - or the recipient's balance would overflow,
///    - or the message sender is 0
///    - or the recipient is 0
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;
    require balanceOf(recipient) + amount < max_uint;
    require e.msg.sender != 0;


    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```

certora

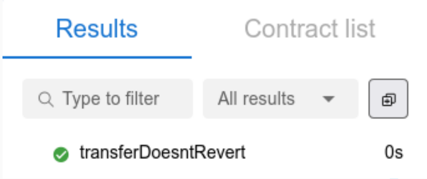
Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///    - the sender doesn't have enough funds
///    - or the message value is nonzero,
///    - or the recipient's balance would overflow,
///    - or the message sender is 0
///    - or the recipient is 0
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;
    require balanceOf(recipient) + amount < max_uint;
    require e.msg.sender != 0;
    require recipient != 0;

    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```

certora

Proving transfer doesn’t revert (liveness rules)

```
//// certora/specs/ERC20.spec

/// Transfer must not revert unless
///     - the sender doesn't have enough funds
///     - or the message value is nonzero,
///     - or the recipient's balance would overflow,
///     - or the message sender is 0
///     - or the recipient is 0
///
/// @title Transfer doesn't revert
rule transferDoesntRevert {
    env e; address recipient; uint amount;

    require balanceOf(e.msg.sender) > amount;
    require e.msg.value == 0;
    require balanceOf(recipient) + amount < max_uint;
    require e.msg.sender != 0;
    require recipient != 0;

    transfer@withrevert(e, recipient, amount);
    assert !lastReverted;
}
```



✓ transferDoesntRevert 0s

(results link)

Summary

- Writing rules is like writing unit tests
  - But you can let the prover choose the values!

Summary

- Writing rules is like writing unit tests
  - But you can let the prover choose the values!
- Use `mathint` variables to avoid overflow in spec

Summary

- Writing rules is like writing unit tests
    - But you can let the prover choose the values!
- Use `mathint` variables to avoid overflow in spec
- Pass `env` as first argument to specify `msg.sender` and other variables
    - Use `envfree` declaration in `methods` block to avoid passing `env`

Summary

- Writing rules is like writing unit tests
  - But you can let the prover choose the values!
- Use `mathint` variables to avoid overflow in spec
- Pass `env` as first argument to specify `msg.sender` and other variables
  - Use `envfree` declaration in `methods` block to avoid passing `env`
- By default, reverting paths are ignored
  - Use `@withrevert` and `lastReverted` to reason about reverting paths
  - Writing “liveness properties” is hard (but possible!)

Exercise (∼15 minutes)

So far (certora/specs/ERC20.spec):

- transferSpec
- transferReverts
- transferDoesntRevert

Exercise:

- Write transferFromSpec
    - ...get it to pass
- Try transferFromReverts
- Try transferFromSucceeds

To run:
sh certora/scripts/verifyERC20.sh

Ask for help!

```
//// contracts/IERC20.sol

/// Interface of the ERC20 standard as defined in the EIP.
interface IERC20 {

    /// Moves `amount` tokens from `sender` to `recipient` using
    /// the allowance mechanism. `amount` is then deducted from
    /// the caller's allowance.
    ///
    function transferFrom(
        address sender,
        address recipient,
        uint256 amount
    ) external returns (bool);


    /// Returns the remaining number of tokens that `spender`
    /// will be allowed to spend on behalf of `owner` through
    /// {transferFrom}.
    ///
    /// This value changes when {approve} or {transferFrom} are
    /// called.
    ///
    function allowance(address owner, address spender)
        external
        view
        returns(uint256);

}
```

CVL: Parametric Rules

certora

Michael George

Stanford, August 2022

Are my funds safe?

So far:

- `transfer` spends the sender's funds
- `transferFrom` reverts if caller's allowance is 0

So if I don't call `transfer` and don't give anyone an allowance, my funds are safe

Are my funds safe?

So far:

- `transfer` spends the sender’s funds
- `transferFrom` reverts if caller’s allowance is 0

So if I don’t call `transfer` and don’t give anyone an allowance, my funds are safe ...right?

Are my funds safe?

So far:

- `transfer` spends the sender’s funds
- `transferFrom` reverts if caller’s allowance is 0

So if I don’t call `transfer` and don’t give anyone an allowance, my funds are safe ...right?

- Do I control my own allowance?

Are my funds safe?

So far:

- `transfer` spends the sender’s funds
- `transferFrom` reverts if caller’s allowance is 0

So if I don’t call `transfer` and don’t give anyone an allowance, my funds are safe ...right?

- Do I control my own allowance?
- Do I control my own balance?

Only token holder can approve (stakeholder rule)

We want to show that the token holder controls their allowances

- Allowances are controlled by `approve`:

```
//// contracts/IERC20.sol

/// Sets `amount` as the allowance of `spender` over the caller's tokens.
function approve(address spender, uint256 amount) external returns (bool);
```

certora

Only token holder can approve (stakeholder rule)

We want to show that the token holder controls their allowances

- Allowances are controlled by `approve`:

```
//// contracts/IERC20.sol

/// Sets `amount` as the allowance of `spender` over the caller's tokens.
function approve(address spender, uint256 amount) external returns (bool);
```

- Maybe check that only the holder can call `approve`?

```
//// certora/specs/ERC20.spec

/// Approve reverts unless called by the owner
rule onlyHolderCanCallApprove {

    address holder; address spender;

    env e; uint256 amount;
    approve@withrevert(e, spender, amount);

    // note: P => Q means "if P then Q" or "P implies Q"
    assert e.msg.sender != holder => lastReverted,
        "approve can only successfully be called by the holder";
}
```

Only token holder can approve (stakeholder rule)

We want to show that the token holder controls their allowances

- Allowances are controlled by approve:

```
//// contracts/IERC20.sol

/// Sets `amount` as the allowance of `spender` over the caller's tokens.
function approve(address spender, uint256 amount) external returns (bool);
```

- Maybe check that only the holder can call approve?

```
//// certora/specs/ERC20.spec

/// Approve reverts unless called by the owner
rule onlyHolderCanCallApprove {

    address holder; address spender;

    env e; uint256 amount;
    approve@withrevert(e, spender, amount);

    // note: P => Q means "if P then Q" or "P implies Q"
    assert e.msg.sender != holder => lastReverted,
        "approve can only successfully be called by the holder";
}
```

- Fails (results link)! Who is the holder?

certora

Only token holder can approve (stakeholder rule)

We want to show that the token holder controls their allowances

- Allowances are controlled by approve:

```
//// contracts/IERC20.sol

/// Sets `amount` as the allowance of `spender` over the caller's tokens.
function approve(address spender, uint256 amount) external returns (bool);
```

- Maybe check that only the holder can call approve?

```
//// certora/specs/ERC20.spec

/// Approve reverts unless called by the owner
rule onlyHolderCanCallApprove {

    address holder; address spender;

    env e; uint256 amount;
    approve@withrevert(e, spender, amount);

    // note: P => Q means "if P then Q" or "P implies Q"
    assert e.msg.sender != holder => lastReverted,
        "approve can only successfully be called by the holder";
}
```

- Fails (results link)! Who is the holder?
  - ...the address whose (outgoing) allowance changes

certora

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if `approve` changes holder’s allowance, then holder called it.

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it.

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;


    env e; uint256 amount;
    approve(e, spender, amount);



    assert allowance_after != allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";



}
```

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it.

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    env e; uint256 amount;
    approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after != allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";



}
```

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it. (passes)

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    env e; uint256 amount;
    approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after != allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";



}
```

certora

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it. (passes)
- if any method changes holder’s balance, then the holder called it.

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    env e; uint256 amount;
    approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after != allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";


}
```

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it. (passes)
- if any method changes holder’s balance, then the holder called it.

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    env e; uint256 amount;
    f(e, args);                        // was: approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after != allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";


}
```

certora

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it. (passes)
- if any method changes holder’s balance, then the holder called it.

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    method f; env e; calldataarg args; // was: env e; uint256 amount;
    f(e, args);                        // was: approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after != allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";


}
```

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder's allowance, then holder called it. (passes)
- if any method changes holder's balance, then the holder called it. (link)

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    method f; env e; calldataarg args; // was: env e; uint256 amount;
    f(e, args);                        // was: approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after != allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";


}
```

certora

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it. (passes)
- if any method changes holder’s balance, then the holder called it. (link)
- if any method increases holder’s balance, then the holder called it

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    method f; env e; calldataarg args; // was: env e; uint256 amount;
    f(e, args);                        // was: approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after != allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";


}
```

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it. (passes)
- if any method changes holder’s balance, then the holder called it. (link)
- if any method increases holder’s balance, then the holder called it

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    method f; env e; calldataarg args; // was: env e; uint256 amount;
    f(e, args);                        // was: approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after > allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";


}
```

certora

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it. (passes)
- if any method changes holder’s balance, then the holder called it. (link)
- if any method increases holder’s balance, then the holder called it (passes)

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    method f; env e; calldataarg args; // was: env e; uint256 amount;
    f(e, args);                        // was: approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after > allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";


}
```

certora

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it. (passes)
- if any method changes holder’s balance, then the holder called it. (link)
- if any method increases holder’s balance, then the holder called it (passes)
    - …and they meant to change the balance

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    method f; env e; calldataarg args; // was: env e; uint256 amount;
    f(e, args);                        // was: approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after > allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";



}
```

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder's allowance, then holder called it. (passes)
- if any method changes holder's balance, then the holder called it. (link)
- if any method increases holder's balance, then the holder called it (passes)
  - ...and they meant to change the balance

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    method f; env e; calldataarg args; // was: env e; uint256 amount;
    f(e, args);                        // was: approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after > allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";
    assert allowance_after > allowance_before =>
        (f.selector == approve(address,uint).selector || f.selector == increaseAllowance(address,uint).selector),
        "only approve and increaseAllowance can increase allowances";
}
```

certora

Only holder can approve, take 2 (variable change rule)

We want to show that the token holder controls their allowances

- if approve changes holder’s allowance, then holder called it. (passes)
- if any method changes holder’s balance, then the holder called it. (link)
- if any method increases holder’s balance, then the holder called it (passes)
  - …and they meant to change the balance (passes)

```
rule onlyHolderCanChangeAllowance {

    address holder; address spender;
    mathint allowance_before = allowance(holder, spender);

    method f; env e; calldataarg args; // was: env e; uint256 amount;
    f(e, args);                        // was: approve(e, spender, amount);

    mathint allowance_after = allowance(holder, spender);

    assert allowance_after > allowance_before => e.msg.sender == holder,
        "addresses other than holder must not affect holder's allowance";
    assert allowance_after > allowance_before =>
        (f.selector == approve(address,uint).selector || f.selector == increaseAllowance(address,uint).selector),
        "only approve and increaseAllowance can increase allowances";
}
```

Summary

- You can use a `method` variable to stand in for an arbitrary method
  - Need to pass an `env` and a `calldataarg` parameter
  - Prover will verify separately on every (external) method in the contract

Summary

- You can use a `method` variable to stand in for an arbitrary method
  - Need to pass an `env` and a `calldataarg` parameter
  - Prover will verify separately on every (external) method in the contract
  - Note: rules using `method` variables are called “parametric rules.”

Summary

- You can use a `method` variable to stand in for an arbitrary method
  - Need to pass an `env` and a `calldataarg` parameter
  - Prover will verify separately on every (external) method in the contract
  - Note: rules using `method` variables are called “parametric rules.”
- You can identify `method` object `f` using `f.selector`

Summary

- You can use a `method` variable to stand in for an arbitrary method
  - Need to pass an `env` and a `calldataarg` parameter
  - Prover will verify separately on every (external) method in the contract
  - Note: rules using `method` variables are called “parametric rules.”
- You can identify `method` object `f` using `f.selector`
- The expression `P => Q` means “if `P` then `Q`” or “`P` implies `Q`”

Summary

- You can use a `method` variable to stand in for an arbitrary method
  - Need to pass an `env` and a `calldataarg` parameter
  - Prover will verify separately on every (external) method in the contract
  - Note: rules using `method` variables are called "parametric rules."
- You can identify `method` object `f` using `f.selector`
- The expression `P => Q` means "if `P` then `Q`" or "`P` implies `Q`"
- Some general rule patterns:
  - Generalizing rules can get good coverage quickly

certora

Summary

- You can use a `method` variable to stand in for an arbitrary method
  - Need to pass an `env` and a `calldataarg` parameter
  - Prover will verify separately on every (external) method in the contract
  - Note: rules using `method` variables are called “parametric rules.”
- You can identify `method` object `f` using `f.selector`
- The expression `P => Q` means “if `P` then `Q`” or “`P` implies `Q`”
- Some general rule patterns:
  - Generalizing rules can get good coverage quickly
  - “Unit test rules”: describe behavior of specific methods
    - e.g. `transferSpec`

certora

Summary

- You can use a `method` variable to stand in for an arbitrary method
  - Need to pass an `env` and a `calldataarg` parameter
  - Prover will verify separately on every (external) method in the contract
  - Note: rules using `method` variables are called “parametric rules.”
- You can identify `method` object `f` using `f.selector`
- The expression `P => Q` means “if `P` then `Q`” or “`P` implies `Q`”
- Some general rule patterns:
  - Generalizing rules can get good coverage quickly
  - “Unit test rules”: describe behavior of specific methods
    - e.g. `transferSpec`
  - “Stakeholder rules”: put yourself in user’s shoes
    - e.g. `onlyHolderCanChangeAllowance`

certora

Summary

- You can use a `method` variable to stand in for an arbitrary method
  - Need to pass an `env` and a `calldataarg` parameter
  - Prover will verify separately on every (external) method in the contract
  - Note: rules using `method` variables are called “parametric rules.”
- You can identify `method` object `f` using `f.selector`
- The expression `P => Q` means “if `P` then `Q`” or “`P` implies `Q`”
- Some general rule patterns:
  - Generalizing rules can get good coverage quickly
  - “Unit test rules”: describe behavior of specific methods
    - e.g. `transferSpec`
  - “Stakeholder rules”: put yourself in user’s shoes
    - e.g. `onlyHolderCanChangeAllowance`
  - “Variable change rules”: describe conditions of variable changes
    - e.g. `onlyHolderCanChangeAllowance`

certora

Summary

- You can use a `method` variable to stand in for an arbitrary method
  - Need to pass an `env` and a `calldataarg` parameter
  - Prover will verify separately on every (external) method in the contract
  - Note: rules using `method` variables are called “parametric rules.”
- You can identify `method` object `f` using `f.selector`
- The expression `P => Q` means “if `P` then `Q`” or “`P` implies `Q`”
- Some general rule patterns:
  - Generalizing rules can get good coverage quickly
  - “Unit test rules”: describe behavior of specific methods
    - e.g. `transferSpec`
  - “Stakeholder rules”: put yourself in user’s shoes
    - e.g. `onlyHolderCanChangeAllowance`
  - “Variable change rules”: describe conditions of variable changes
    - e.g. `onlyHolderCanChangeAllowance`
  - More on rule patterns tomorrow!

Exercise (∼15 minutes)

We just wrote rules for allowance changes

- In certora/specs/ERC20.spec
- If allowance increases, then the sender was the holder, and the method was appropriate

Now, write rules for balance changes

- In certora/specs/ERC20.spec
- If my balance goes down, what should I know?

Invariants

certora

Michael George

Stanford, August 2022

Invariants

What is an invariant?

Invariants

What is an invariant?

- Something that doesn't change over time

Invariants

What is an invariant?

- Something that doesn’t change over time
- A property of the state (storage) that should be true between transactions

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)

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:

Things that are not invariants:

certora

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:

- The balance of the zero address is zero

Things that are invariants:

Things that are not invariants:

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:

- The balance of the zero address is zero

Things that are not invariants:

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:

- The total supply is the sum of all user balances

Things that are invariants:

- The balance of the zero address is zero

Things that are not invariants:

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:

- The balance of the zero address is zero
- The total supply is the sum of all user balances

Things that are not invariants:

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:

- `transferFrom` reverts if the sender’s allowance is 0

Things that are invariants:

- The balance of the zero address is zero
- The total supply is the sum of all user balances

Things that are not invariants:

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:

- The balance of the zero address is zero
- The total supply is the sum of all user balances

Things that are not invariants:

- `transferFrom` reverts if the sender's allowance is 0

certora

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:

- Assets exceed liabilities (solvency)

Things that are invariants:

- The balance of the zero address is zero
- The total supply is the sum of all user balances

Things that are not invariants:

- `transferFrom` reverts if the sender's allowance is 0

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:

- 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:

- `transferFrom` reverts if the sender's allowance is 0

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:

- A user's rewards can only increase

Things that are invariants:

- 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:

- `transferFrom` reverts if the sender's allowance is 0

certora

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:

- 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:

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

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:

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

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
```

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

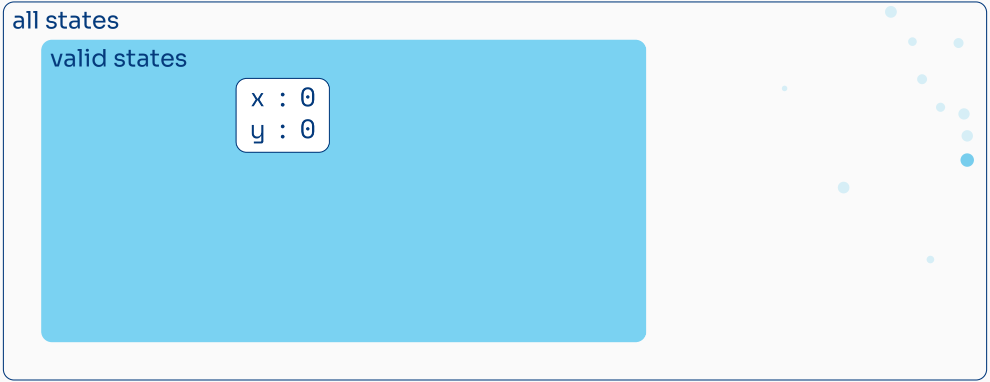
- Invariant: $x \geq y$

all states

valid states

Checking invariants

- Invariant: `x ≥ y`



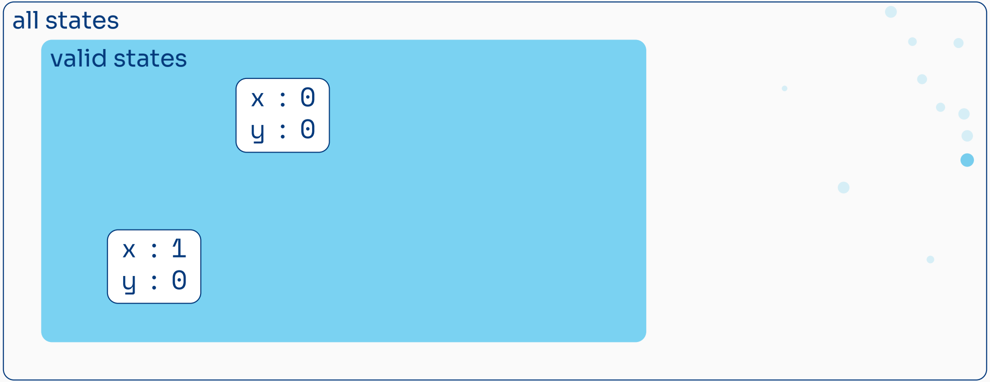
all states

valid states

x : 0
y : 0

Checking invariants

- Invariant: `x ≥ y`



all states

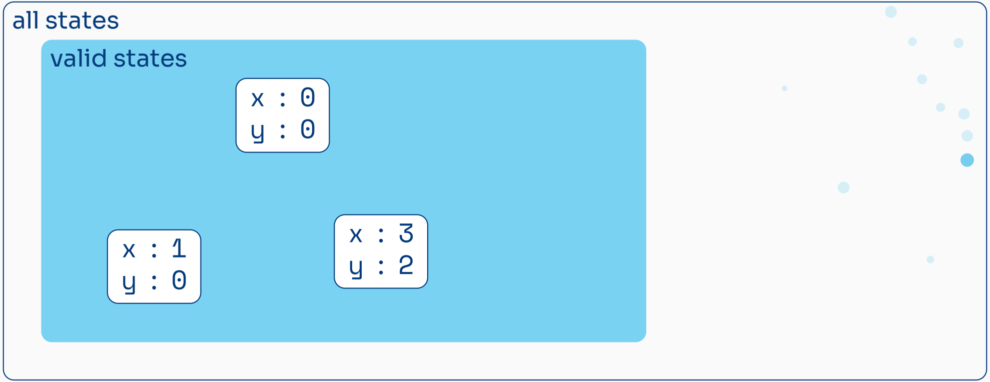
valid states

```
x : 0
y : 0
```

```
x : 1
y : 0
```

Checking invariants

- Invariant: `x ≥ y`



all states

valid states

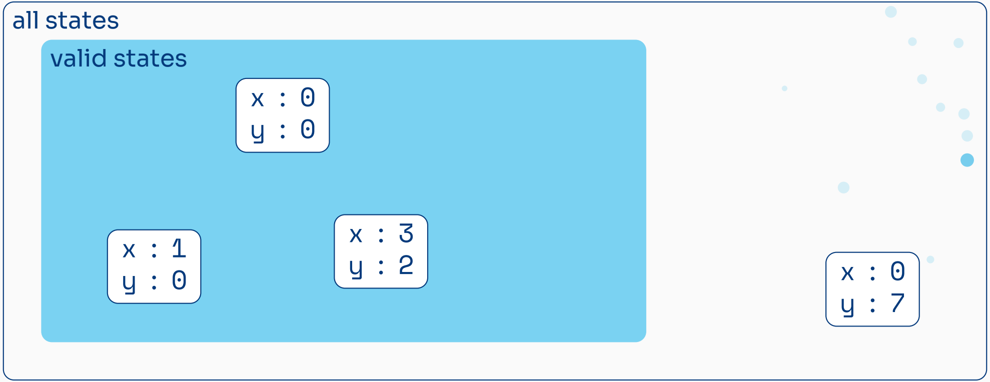
x : 0
y : 0

x : 1
y : 0

x : 3
y : 2

Checking invariants

- Invariant: `x ≥ y`



all states

valid states

x : 0
y : 0

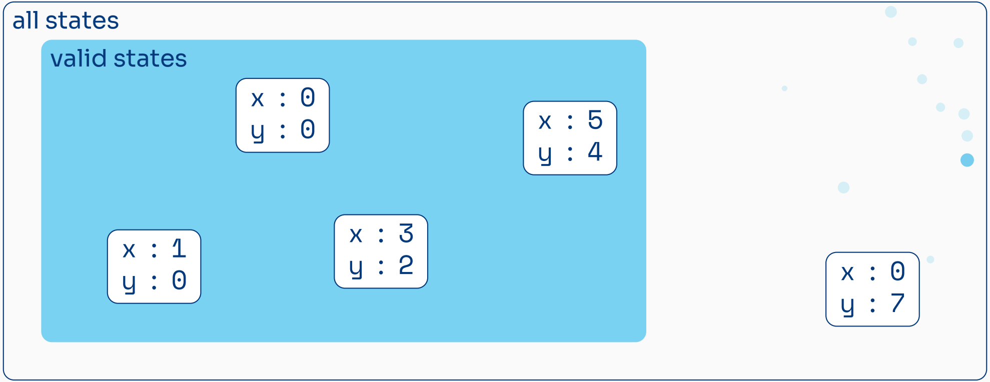
x : 1
y : 0

x : 3
y : 2

x : 0
y : 7

Checking invariants

- Invariant: `x ≥ y`



all states

valid states

x : 0
y : 0

x : 5
y : 4

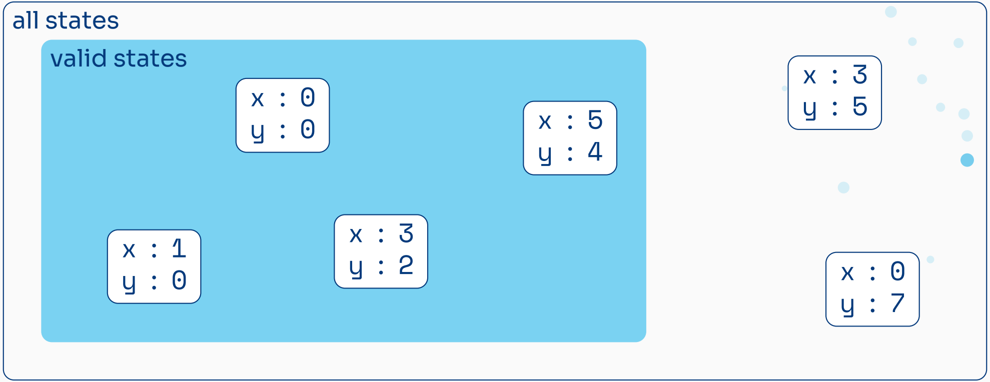
x : 1
y : 0

x : 3
y : 2

x : 0
y : 7

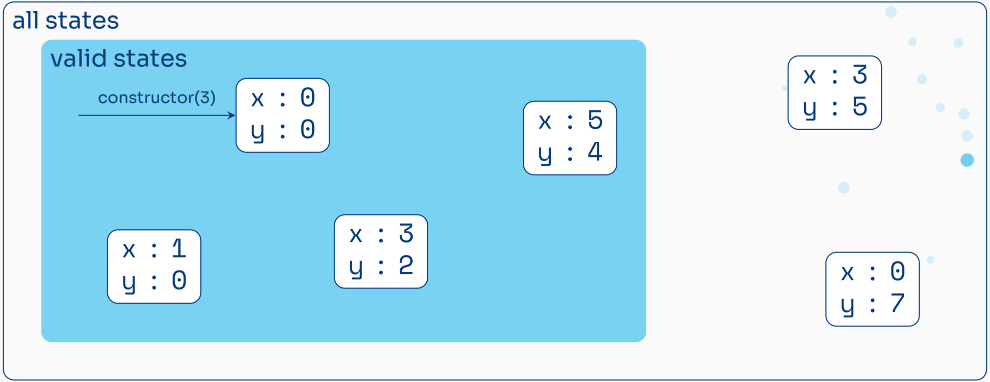
Checking invariants

- Invariant: `x ≥ y`



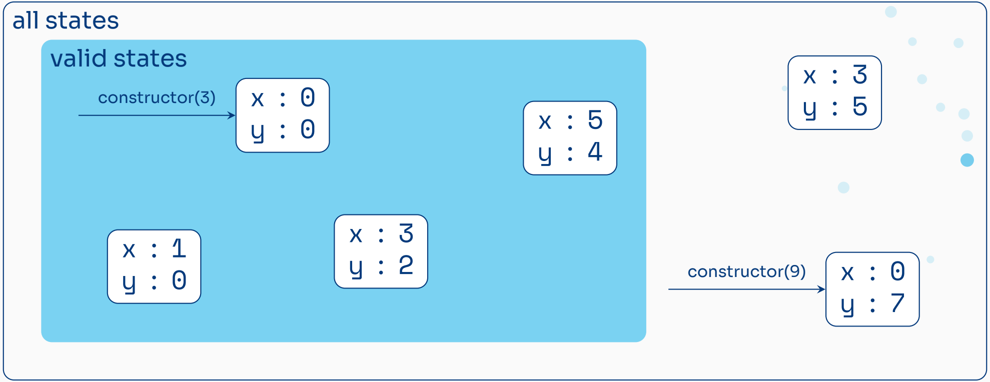
Checking invariants

- Invariant: `x ≥ y`



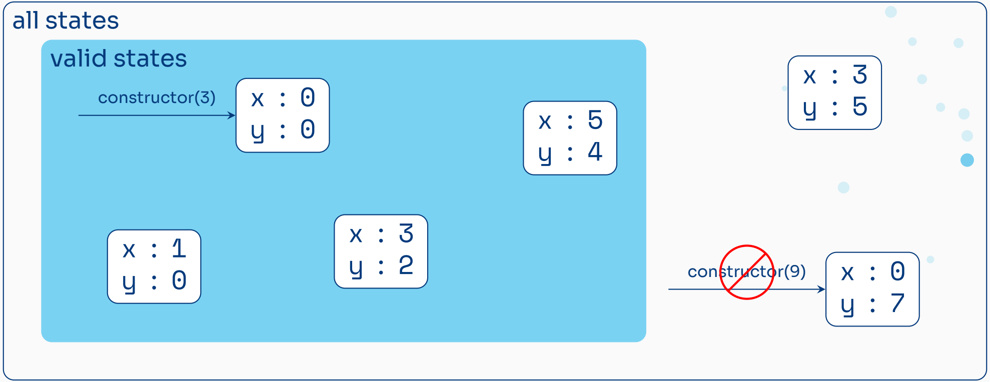
Checking invariants

- Invariant: `x ≥ y`



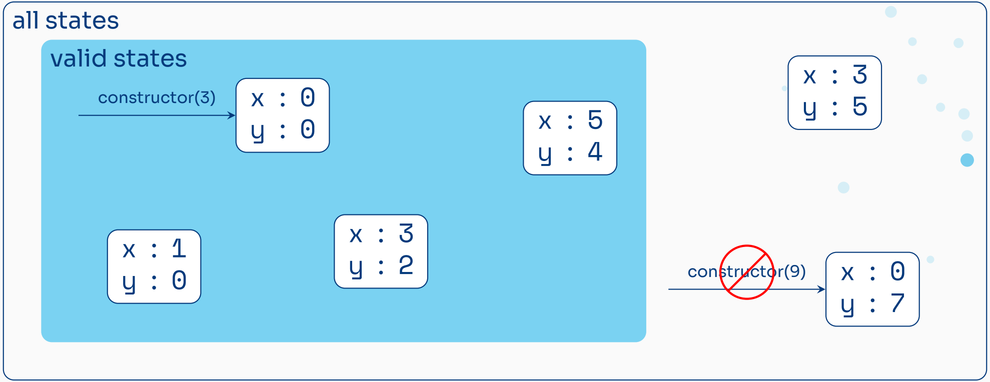
Checking invariants

- Invariant: `x ≥ y`



Checking invariants

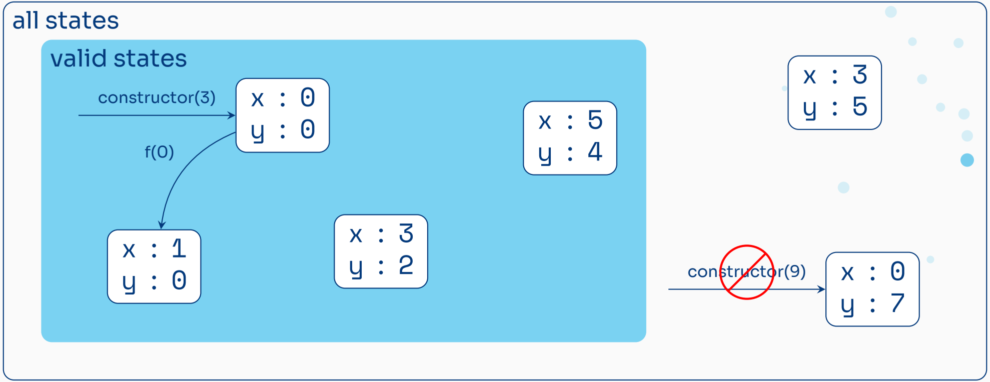
- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid

Checking invariants

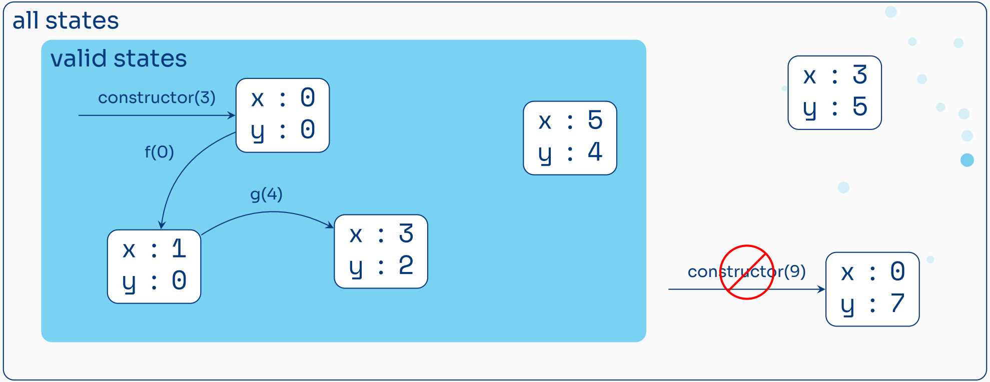
- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid

Checking invariants

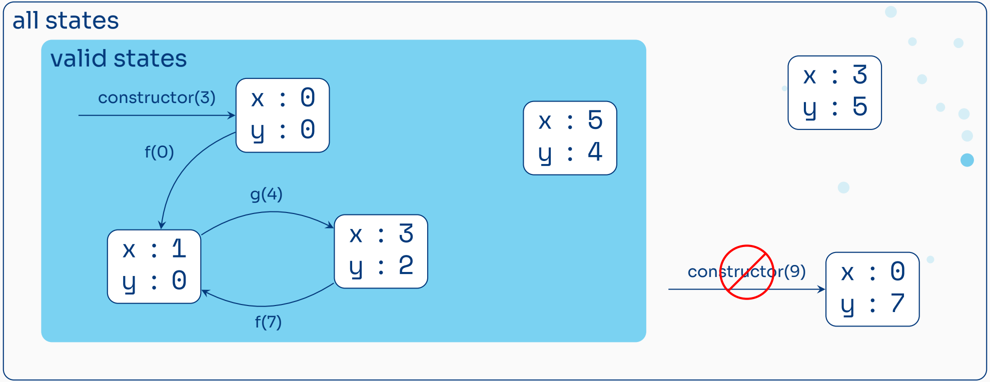
- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid

Checking invariants

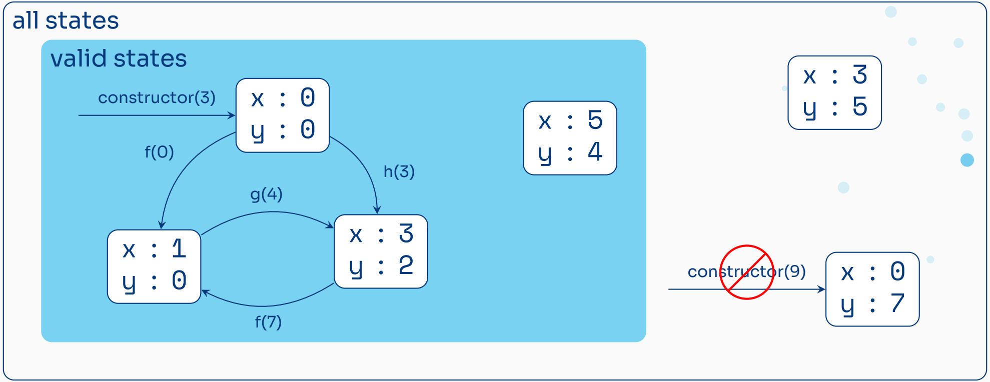
- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid

Checking invariants

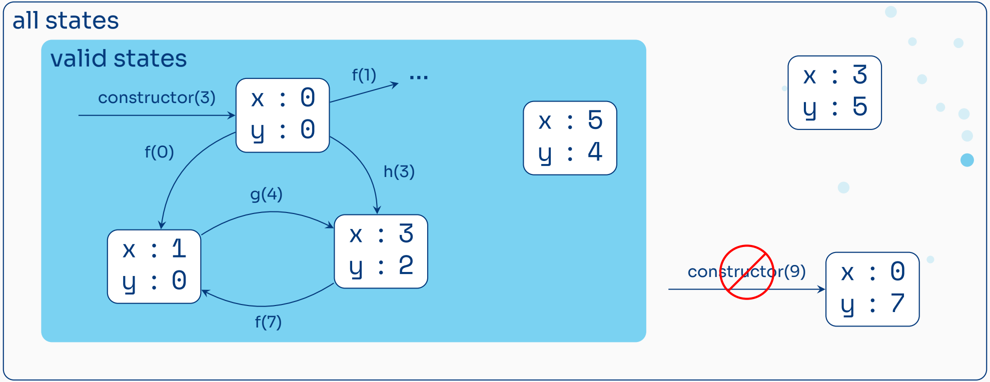
- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid

Checking invariants

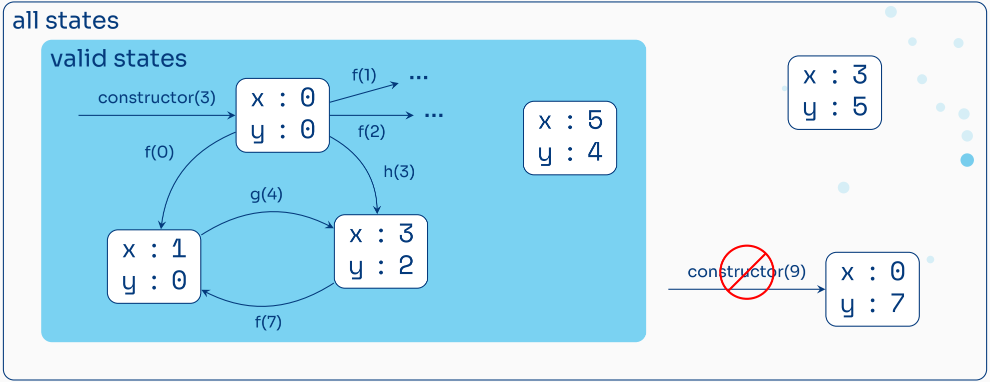
- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid

Checking invariants

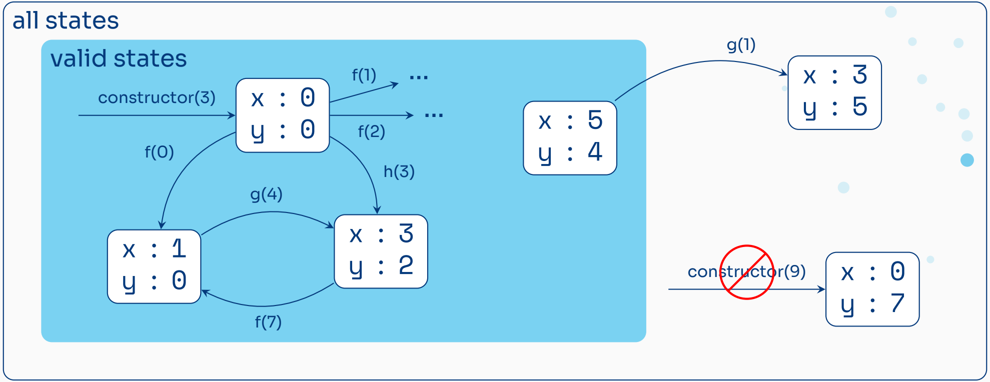
- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid

Checking invariants

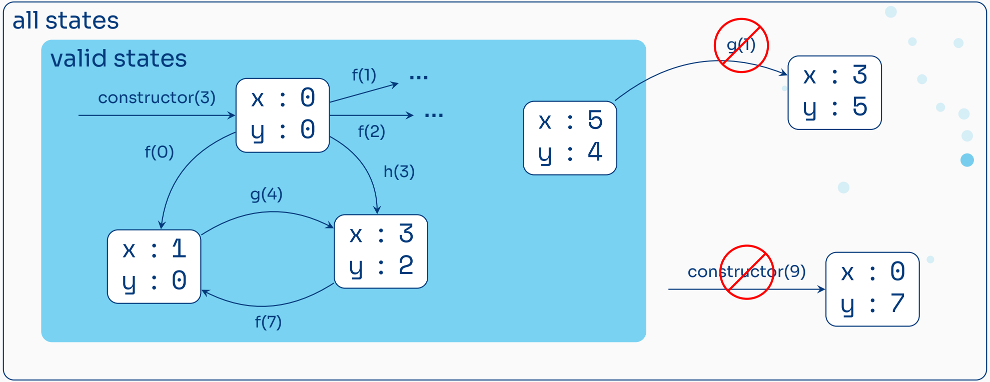
- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid

Checking invariants

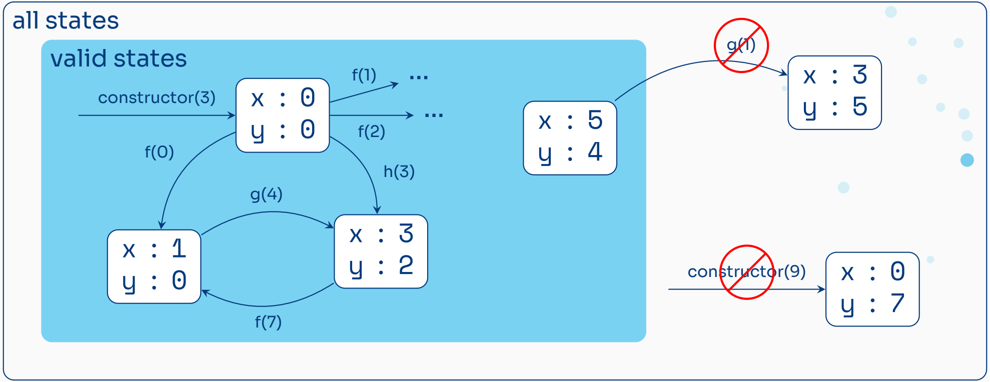
- Invariant: `x ≥ y`



- Need to check that initial state (after any constructor call) is valid

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
```

Verifying an invariant

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

(results link)

certora

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)

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)

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

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!

certora

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

certora

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

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?

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!

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
```

Solution walkthrough

Goal: player 2 never gets the ball

- First attempt:

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

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

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;
    }
}
```

certora

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)

Third attempt: rule out more bad cases

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

Third attempt: rule out more bad cases

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

Passes! (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

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?

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;
}
```

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)!

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?

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!

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

Fourth attempt: strengthening the invariant

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

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
```

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.

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

Returning to original goal

- We wanted to prove `ballPosition() != 2`

Returning to original goal

- We wanted to prove `ballPosition() != 2`
- Instead we proved `ballPosition() == 1 || ballPosition() == 3`

Returning to original goal

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

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
```

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
    }
}
```

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)

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

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!

Back to ERC20

certora

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()
```

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!

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()
  ```

certora

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!

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

certora

Summary

Things we covered in this session

Summary

Things we covered in this session

- Invariants are properties of the state that don’t change over time

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)

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!

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
    - `requireInvariant`s
    - 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!

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
    - `requireInvariant`s
    - 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

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
    - `requireInvariant`s
    - 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

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)

certora

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

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

Ghosts

certora

Michael George

Stanford, August 2022

A (much) stronger invariant

Before the break:

- Tried to show that each user balance is at most the total supply

Now:

- We’ll show that the total supply is the sum of all user balances

A (much) stronger invariant

Before the break:

- Tried to show that each user balance is at most the total supply

Now:

- We'll show that the total supply is the sum of all user balances

$$\texttt{totalSupply()} = \sum_{\texttt{a} \in \texttt{address}} \texttt{balanceOf}(a)$$

certora

A (much) stronger invariant

Before the break:

- Tried to show that each user balance is at most the total supply

Now:

- We’ll show that the total supply is the sum of all user balances

$$\texttt{totalSupply()} = \sum_{a \in \texttt{address}} \texttt{balanceOf(a)}$$

- It’s hard to track infinite sum
  - we’ll track changes to balances instead

A (much) stronger invariant

Before the break:

- Tried to show that each user balance is at most the total supply

Now:

- We’ll show that the total supply is the sum of all user balances

$$\texttt{totalSupply()} = \sum_{a \in \texttt{address}} \texttt{balanceOf(a)}$$

- It’s hard to track infinite sum
  - we’ll track changes to balances instead

Hooks

A CVL hook allows running CVL code when the contract updates storage

Hooks

A CVL hook allows running CVL code when the contract updates storage

- Syntax:

```
hook Sstore <pattern> <new variable> (<old variable>) STORAGE {
    <body>
}
```

Hooks

A CVL hook allows running CVL code when the contract updates storage

- Syntax:

```
hook Sstore <pattern> <new variable> (<old variable>) STORAGE {
    <body>
}
```

- Example:

```
hook Sstore _balances[KEY address a] uint new_value (uint old_value) STORAGE {
    ...
}
```

Hooks

A CVL hook allows running CVL code when the contract updates storage

- Syntax:

```
hook Sstore <pattern> <new variable> (<old variable>) STORAGE {
    <body>
}
```

- Example:

```
hook Sstore _balances[KEY address a] uint new_value (uint old_value) STORAGE {
    ...
}
```

- Pattern is a field followed by any number of:
  - array lookups (using `[INDEX <type> <name>]`),
  - mapping lookups (using `[KEY <type> <name>]`),
  - struct field lookups (using `.field`)

Hooks

A CVL hook allows running CVL code when the contract updates storage

- Syntax:

```
hook Sstore <pattern> <new variable> (<old variable>) STORAGE {
    <body>
}
```

- Example:

```
hook Sstore _balances[KEY address a] uint new_value (uint old_value) STORAGE {
    ...
}
```

- Pattern is a field followed by any number of:
  - array lookups (using `[INDEX <type> <name>]`),
  - mapping lookups (using `[KEY <type> <name>]`),
  - struct field lookups (using `.field`)

- Hook can update our tracked sum of balances

Ghosts

A ghost variable is an additional variable that doesn’t exist in the contract

- Primarily useful for keeping track of changes from hooks

Ghosts

A ghost variable is an additional variable that doesn’t exist in the contract

- Primarily useful for keeping track of changes from hooks

Example:

```
ghost mathint sum_of_balances;
```

Ghosts

A ghost variable is an additional variable that doesn’t exist in the contract

- Primarily useful for keeping track of changes from hooks

Example:

```
ghost mathint sum_of_balances;
```

You can also declare ghost mappings:

```
ghost mapping(address => mapping(address => uint256)) balances_by_token;
```

Ghosts

A ghost variable is an additional variable that doesn’t exist in the contract

- Primarily useful for keeping track of changes from hooks

Example:

```
ghost mathint sum_of_balances;
```

You can also declare ghost mappings:

```
ghost mapping(address => mapping(address => uint256)) balances_by_token;
```

Prover considers every possible value of ghost (just like storage)

Putting ghost and hook together

Example (results link):

```
ghost mathint sum_of_balances;



hook Sstore _balances[KEY address a] uint new_value (uint old_value) STORAGE {
    // when balance changes, update ghost
    sum_of_balances = sum_of_balances + new_value - old_value;
}

invariant totalSupplyIsSumOfBalances()
    totalSupply() == sum_of_balances
```

Putting ghost and hook together

Example (results link):

```
ghost mathint sum_of_balances;



hook Sstore _balances[KEY address a] uint new_value (uint old_value) STORAGE {
    // when balance changes, update ghost
    sum_of_balances = sum_of_balances + new_value - old_value;
}

invariant totalSupplyIsSumOfBalances()
    totalSupply() == sum_of_balances
```

Rule passes on preservation but fails on initialization

- Prover chooses non-zero initial value for the ghost

Putting ghost and hook together

Example (results link):

```
ghost mathint sum_of_balances {
    init_state axiom sum_of_balances == 0;
}

hook Sstore _balances[KEY address a] uint new_value (uint old_value) STORAGE {
    // when balance changes, update ghost
    sum_of_balances = sum_of_balances + new_value - old_value;
}

invariant totalSupplyIsSumOfBalances()
    totalSupply() == sum_of_balances
```

Rule passes on preservation but fails on initialization

- Prover chooses non-zero initial value for the ghost

Putting ghost and hook together

Example (results link):

```
hook Sstore _balances[KEY address a] uint new_value (uint old_value) STORAGE {
    // when balance changes, update ghost
    sum_of_balances = sum_of_balances + new_value - old_value;
}

invariant totalSupplyIsSumOfBalances()
    totalSupply() == sum_of_balances
```

Rule passes on preservation but fails on initialization

- Prover chooses non-zero initial value for the ghost

Initial state axiom tells prover to make assumptions about the intial value of the ghost (before the constructor)

Exercise

- Create a ghost to track the number of changes to users’ balances
- Use it to prove that no method changes more than two balances

Hyperproperties

Michael George

Stanford, August 2022

Comparing hypothetical executions

Often, we want to say that two operations have the same effects

Comparing hypothetical executions

Often, we want to say that two operations have the same effects

- Two small deposits are the same as one big deposit

Comparing hypothetical executions

Often, we want to say that two operations have the same effects

- Two small deposits are the same as one big deposit
- Adding permissions doesn’t cause reverts

Comparing hypothetical executions

Often, we want to say that two operations have the same effects

- Two small deposits are the same as one big deposit
- Adding permissions doesn’t cause reverts
- Staking more earns more

Comparing hypothetical executions

Often, we want to say that two operations have the same effects

- Two small deposits are the same as one big deposit
- Adding permissions doesn't cause reverts
- Staking more earns more
- Staking longer earns more

Comparing hypothetical executions

Often, we want to say that two operations have the same effects

- Two small deposits are the same as one big deposit
- Adding permissions doesn’t cause reverts
- Staking more earns more
- Staking longer earns more

CVL allows saving and restoring the state of the world

Comparing hypothetical executions

Often, we want to say that two operations have the same effects

- Two small deposits are the same as one big deposit
- Adding permissions doesn’t cause reverts
- Staking more earns more
- Staking longer earns more

CVL allows saving and restoring the state of the world

- `storage` type represents a snapshot of storage

Comparing hypothetical executions

Often, we want to say that two operations have the same effects

- Two small deposits are the same as one big deposit
- Adding permissions doesn’t cause reverts
- Staking more earns more
- Staking longer earns more

CVL allows saving and restoring the state of the world

- `storage` type represents a snapshot of storage
- `lastStorage` gives the current state of storage

Comparing hypothetical executions

Often, we want to say that two operations have the same effects

- Two small deposits are the same as one big deposit
- Adding permissions doesn't cause reverts
- Staking more earns more
- Staking longer earns more

CVL allows saving and restoring the state of the world

- `storage` type represents a snapshot of storage
- `lastStorage` gives the current state of storage
- `f(...) at s` resets the storage before executing `f`

Example

- Want to show that transferring a and the b is the same as transferring a + b

```
//// certora/specs/ERC20.spec

/// transferring `a` tokens and then then `b` tokens has the same effect as
/// transferring `a+b` tokens
rule transferFromAdditive {
    address sender; address recipient;
    uint amount_a; uint amount_b;

    storage init = lastStorage;                                   // save storage

    transferFrom(sender, recipient, amount_a);
    transferFrom(sender, recipient, amount_b);

    mathint balance_sender_1 = balanceOf(sender);
    mathint balance_recip_1  = balanceOf(recipient);

    transferFrom(sender, recipient, amount_a + amount_b) at init; // restore storage

    mathint balance_sender_2    = balanceOf(sender);
    mathint balance_recipient_2 = balanceOf(recipient);

    assert balance_sender_1 == balance_sender_2,
        "two small transfers must change the sender's balance by the same amount as one large transfer";

    assert balance_recip_1 == balance_recip_2,
        "two small transfers must change the recipient's balance by the same amount as one large transfer";
}
```

The Art and Science of Designing Specifications

certora

Types of properties

Types of properties

So far: how to write specs

Types of properties

So far: how to write specs

Now: what specs to write?

Types of properties

So far: how to write specs

Now: what specs to write?

When designing specifications, it helps to work systematically

- Unit-test-style rules
- Variable relationships and changes
- State transition diagrams
- Stakeholder rules
- High-level properties

Unit-test style rules

- Public functions and interfaces should have documentation
  - Describe what their arguments are
  - Describe what effects they should have
  - Describe what they should return
  - Describe when they should revert

Unit-test style rules

- Public functions and interfaces should have documentation
  - Describe what their arguments are
  - Describe what effects they should have
  - Describe what they should return
  - Describe when they should revert

- This documentation can usually be turned directly into specs
  - You can write one or more rules for each method
  - We call these “unit-test style rules”
  - Example: transfer decreases sender’s balance by amount

Unit-test style rules

- Public functions and interfaces should have documentation
  - Describe what their arguments are
  - Describe what effects they should have
  - Describe what they should return
  - Describe when they should revert
- This documentation can usually be turned directly into specs
  - You can write one or more rules for each method
  - We call these “unit-test style rules”
  - Example: transfer decreases sender’s balance by amount
- Note: you can get a list of public functions from the Prover (example)

Unit-test style rules

- Public functions and interfaces should have documentation
  - Describe what their arguments are
  - Describe what effects they should have
  - Describe what they should return
  - Describe when they should revert
- This documentation can usually be turned directly into specs
  - You can write one or more rules for each method
  - We call these “unit-test style rules”
  - Example: transfer decreases sender’s balance by amount
- Note: you can get a list of public functions from the Prover (example)
- In practice, the documentation is often incomplete
  - Think about the documentation you’d write
  - Maybe submit a PR!

Variable relationships and changes

Variable relationships

- For each pair of variables, ask “how are they related”?
- Each relationship can be written as an invariant
- Include related contracts!

Variable relationships and changes

Variable relationships

- For each pair of variables, ask “how are they related”?
- Each relationship can be written as an invariant
- Include related contracts!

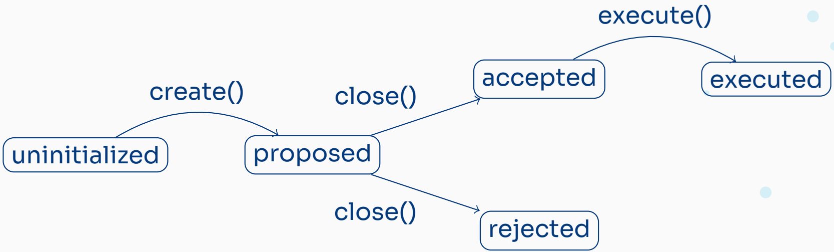
Variable changes

- For each variable, ask “how can it change, and when?”
- Each variable has one or more parametric rules:

```
rule variableChange {
    mathint value_before = getValue();

    method f; env e; calldataarg args;
    f(e,args);

    mathint value_after = getValue();

    assert value_before != value_after => ...;
}
```

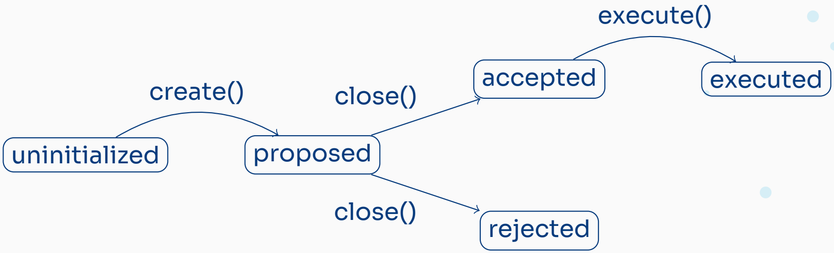
State transition diagrams

Often contracts have a natural “flow-chart” feel:



State transition diagrams

Often contracts have a natural “flow-chart” feel:



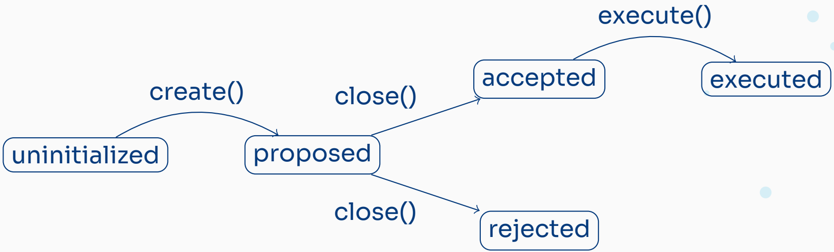
These can naturally be turned into rules:

- Define properties of each state

```
definition accepted_state (env e) returns bool =
    initialized() && executable() != 0 && for() > against() && e.block.timestamp > deadline()
```

State transition diagrams

Often contracts have a natural “flow-chart” feel:



These can naturally be turned into rules:

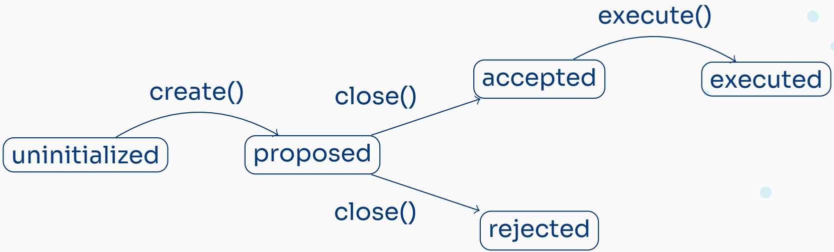
- Define properties of each state

```
definition accepted_state (env e) returns bool =
    initialized() && executable() != 0 && for() > against() && e.block.timestamp > deadline()
```

- Invariant: contract is always in one and only one state

State transition diagrams

Often contracts have a natural “flow-chart” feel:



execute()

accepted

executed

create()

close()

uninitialized

proposed

close()

rejected

These can naturally be turned into rules:

- Define properties of each state

```
definition accepted_state (env e) returns bool =
    initialized() && executable() != 0 && for() > against() && e.block.timestamp > deadline()
```

- Invariant: contract is always in one and only one state
- Each transition can have one or more rules, like variable changes

Stakeholder rules

Think about what can go wrong from stakeholders' perspectives

- User: I deposit funds and can't get them back
- Bank: Someone removes all the funds

Stakeholder rules

Think about what can go wrong from stakeholders’ perspectives

- User: I deposit funds and can’t get them back
- Bank: Someone removes all the funds

Each “user (horror) story” can be turned into properties

Stakeholder rules

Think about what can go wrong from stakeholders' perspectives

- User: I deposit funds and can't get them back
- Bank: Someone removes all the funds

Each "user (horror) story" can be turned into properties

Often multiple rules: e.g. to show "after deposit I can reclaim funds"

- If I deposit, I get a balance
- My balance doesn't go down unless I withdraw
- I can always withdraw without revert
- When I withdraw, the contract transfers tokens to me

High-level properties

There are some simple properties that can often get good coverage

High-level properties

There are some simple properties that can often get good coverage

- If this goes up, that goes up (correlation)

High-level properties

There are some simple properties that can often get good coverage

- If this goes up, that goes up (correlation)
- If this is zero, that is zero

High-level properties

There are some simple properties that can often get good coverage

- If this goes up, that goes up (correlation)
- If this is zero, that is zero
- Two small operations are the same as one big operation (additivity)

High-level properties

There are some simple properties that can often get good coverage

- If this goes up, that goes up (correlation)
- If this is zero, that is zero
- Two small operations are the same as one big operation (additivity)
- Different ways to do the same thing have the same effect

High-level properties

There are some simple properties that can often get good coverage

- If this goes up, that goes up (correlation)
- If this is zero, that is zero
- Two small operations are the same as one big operation (additivity)
- Different ways to do the same thing have the same effect

Sometimes, more abstract properties are useful

- Get good coverage quickly
- Help us think in a different way, avoiding spec bugs

Summary

When designing specifications, it helps to work systematically

- Unit-test-style rules
  - Describe the expected behavior of each function

Summary

When designing specifications, it helps to work systematically

- Unit-test-style rules
  - Describe the expected behavior of each function
- Variable relationships and changes
  - Describe the relationships between pairs of variables
  - Describe the conditions when variables change

Summary

When designing specifications, it helps to work systematically

- Unit-test-style rules
    - Describe the expected behavior of each function
- Variable relationships and changes
    - Describe the relationships between pairs of variables
    - Describe the conditions when variables change
- State transition diagrams
    - Identify parts of the contract that transition from state to state
    - Check that contract is always in exactly one state
    - Describe conditions when transitions happen

Summary

When designing specifications, it helps to work systematically

- Unit-test-style rules
  - Describe the expected behavior of each function
- Variable relationships and changes
  - Describe the relationships between pairs of variables
  - Describe the conditions when variables change
- State transition diagrams
  - Identify parts of the contract that transition from state to state
  - Check that contract is always in exactly one state
  - Describe conditions when transitions happen
- Stakeholder rules
  - Think about what can go wrong
  - Look at your advertising

Summary

When designing specifications, it helps to work systematically

- Unit-test-style rules
  - Describe the expected behavior of each function
- Variable relationships and changes
  - Describe the relationships between pairs of variables
  - Describe the conditions when variables change
- State transition diagrams
  - Identify parts of the contract that transition from state to state
  - Check that contract is always in exactly one state
  - Describe conditions when transitions happen
- Stakeholder rules
  - Think about what can go wrong
  - Look at your advertising
- High-level properties
  - Think abstractly about your functions and their relationships

AAVE Token Example

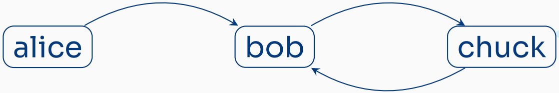
Voting and delegation

The AAVE token is used for voting on proposals

- The more tokens you hold, the more votes you get

You can delegate your vote to another address:

- Delegation is all-or-nothing
- You can't redelegate tokens



Delegation:	alice	bob	chuck
Token balance:	10	7	5
Voting power:	0	15	7

A few more details

- The token manages two types of voting power: `VOTING` and `PROPOSITION`

- The contract supports “meta-delegation”
  - Allows delegation for someone other than `msg.sender`
  - Requires a digital certificate

- The contract is also an ERC20

Exercise: write (English) properties for governance

1. Fetch the code: in the `Examples` repo,
   - `git pull`
   - `git submodule update --init`
   - Alternately, get directly at `https://github.com/Certora/aave-token-v3`
2. Review the interfaces
   - Main interface is in `src/interfaces/IGovernancePowerDelegationToken.sol`
   - The token also implements the ERC20 interface
3. Start writing down properties!
   - `https://bit.ly/certora-stanford/`

certora