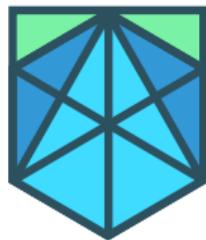


DAY 2: DESIGNING SPECIFICATIONS



MICHAEL GEORGE
CERTORA

Certora AAVE Training

DAY 1 RECAP

Prover overview

- ▶ Rules are like small programs that call contract functions
- ▶ May have undefined variables
- ▶ Prover considers every possible combination of value
- ▶ Reports counterexamples if there are any

Basic Prover usage

- ▶ `require`, `assert`, `mathint`, `env`, `envfree`, `method`, `calldataarg`, `withrevert`, and `lastReverted`
- ▶ navigating the call trace and understanding counterexamples

Unit-test style rules

- ▶ e.g. “`transfer` must increase recipient’s balance by amount”

Parametric rules

- ▶ e.g. “`allowance(owner, spender)` can only be increased by `owner`”

DAY 2 PLAN

Part 1: More CVL features

- ▶ Goal: totalSupply is sum of balances
- ▶ Features: definitions, invariants, ghosts, hooks

Short break

Part 2: Designing specifications

- ▶ Rule coverage
- ▶ Rule design patterns
 - ▶ Unit tests, variable changes, variable relationships, state-transition diagrams, risk assessment, mathematical properties

Demo

DAY 2 PLAN

Part 1: More CVL features

- ▶ Goal: totalSupply is sum of balances
- ▶ Features: definitions, invariants, ghosts, hooks

Short break

Part 2: Designing specifications

- ▶ Rule coverage
- ▶ Rule design patterns
 - ▶ Unit tests, variable changes, variable relationships, state-transition diagrams, risk assessment, mathematical properties

ASSESSING RULE COVERAGE: WHEN ARE YOU DONE?

Code coverage measures quality of **unit tests**

- ▶ number of lines executed during unit tests
- ▶ 100% coverage is not a guarantee of correctness
 - ▶ ...but it gives some assurance that test suite is comprehensive

What's the right metric for quality of **specifications**?

- ▶ Not code coverage: any parametric rule instantly gives 100% code coverage

Certora's QA process:

- ▶ Systematic rule design (today's topic)
- ▶ Bug injection
 - ▶ Insert bugs into contracts, check that the rules catch them
 - ▶ Best if done by someone other than rule writer
- ▶ (Coming soon) randomized mutation testing
 - ▶ Automatically inject simple bugs
 - ▶ E.g. swap argument order, remove modifiers, drop `require` statements

TYPES OF RULES

- ▶ [Most concrete] “Unit test style” / method specification properties
 - ▶ Encode the method documentation
 - ▶ Consider successful cases and revert conditions
- ▶ Variable changes
 - ▶ For each variable / getter, ask when and how it should change
 - ▶ Describe quality of changes and methods that can change
- ▶ Variable relationships
 - ▶ Ask what variables are related to each other and how
- ▶ State-transition diagrams
 - ▶ Ask how a contract (or part of the state) can evolve
- ▶ Risk assessment
 - ▶ Take perspective of different stakeholders
- ▶ [Most abstract] Mathematical properties
 - ▶ Abstract away from details [e.g. monotonicity, additivity, commutativity]

UNIT-TEST STYLE RULES (SINGLE-METHOD SPECS)

Writing method spec rules:

- ▶ Describe expected output
- ▶ Describe expected state changes
- ▶ Describe revert conditions

Example from demo: `transferSpec` and `transferRevertSpec`

```
rule methodSpec {
  env e; address ...;           // declare arguments
  require ...;                 // preconditions

  mathint value_before = ...; // saved pre-state

  mathint result = f(args);    // call method

  assert ...;                  // check post-state
}
```

Good coverage: a spec for every method

- ▶ Except for fiat methods like `balanceOf`

VARIABLE CHANGES

Writing variable change rules:

- ▶ Describe how a variable should evolve (e.g. only increasing, ...)
- ▶ Describe methods that are allowed to change a variable

Example from demo: `onlyOwnerCanDecreaseBalance`

```
rule varChanges {
  mathint value_before = var();           // save old value

  env e; method f; calldataarg args;     // call arbitrary method
  f(e, args);

  mathint value_after = var();           // save new value

  assert value_before != value_after => // if changed then ...
    ...,
    "var must only change if ...";

  assert value_before != value_after => // only changed by ...
    f.selector == mint(uint).selector,
    "only mint can change var";
}
```

Good coverage: changes for every variable

- ▶ Except unconstrained variables

VARIABLE RELATIONSHIPS (INVARIANTS)

Writing variable relationships:

- ▶ Identify groups of related variables
 - ▶ Including variables of related contracts (e.g. `underlying.balanceOf(currentContract)`)
- ▶ Ask “how could I tell if state is valid by looking at these variables?”
- ▶ Often good to write these early so you can use `requireInvariant`

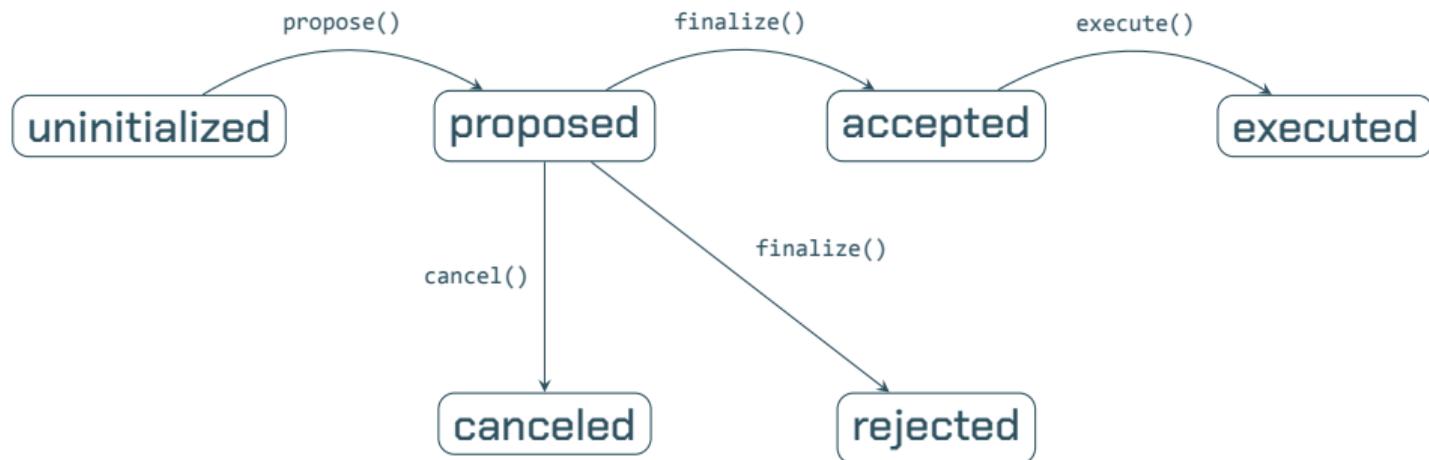
Example from demo: `totalSupplyBounded`

```
invariant x_and_y_correlation()  
...           // describe desired relationship
```

Good coverage: groups of related variables have invariants

STATE-TRANSITION SYSTEMS

Example: Governance system



Definitions:

- ▶ **uninitialized:** `id() == 0`
- ▶ **executed:**
`isClosed() && votesFor() > threshold()`
- ▶ ...

Properties:

- ▶ Can only vote in proposed state
- ▶ Once canceled always canceled
- ▶ ...

STATE-TRANSITION RULES

Writing state-transition rules:

```
definition state1() returns bool = ...;           // define states
definition state2() returns bool = ...;
definition state3() returns bool = ...;

invariant inSomeState()                          // ensure states cover possibilities
  state1() || state2() || state3()

invariant inOneState()                           // if necessary, check state disjointness
  (state1() => !state2() && !state3())
  && ...

rule state1ToState2 {                             // check state transitions
  ...
  require state1_before && state2_after;
  assert ...;
}

invariant varValid()                             // use definitions in rules and invariants
  state1() => ...

rule methodSpec {
  ...
  if (state1()) {
    ...
  } else if (state2()) {
    ...
  } else {
    ...
  }
}
```

RISK ASSESSMENT

Using risk assessment to define rules:

- ▶ Identify stakeholders
 - ▶ e.g. traders, liquidity providers, owners, voters, ...
- ▶ Ask what would make them most unhappy
 - ▶ e.g. fees too high, insolvency, DOS, front-running, ...
- ▶ Rule those things out

Example:

```
definition assets() returns uint256 = underlyingBalance(currentContract);
definition liabilities() returns uint256 = totalSupply() * conversion_factor();

invariant solvency()
  assets() >= liabilities()
```

Good coverage: All stakeholders are happy

MATHEMATICAL ABSTRACTIONS

Abstract properties of functions as mathematical functions can reveal bugs

- ▶ **Monotonicity:** a function is only increasing or only decreasing
- ▶ **Correlation:** if one function increases, so does another
- ▶ **Commutativity:** it doesn't matter which order two operations happen in
- ▶ **Additivity:** the effect of two operations is the sum of their individual effects

Example:

```
rule depositAdditivity {
  uint amountA; uint amountB; env e;

  storage init = lastStorage; // save state of storage for replay

  deposit(e, amountA); // deposit two smaller amounts
  deposit(e, amountB);

  uint separate_balance = balanceOf(e.msg.sender); // save resulting balance

  deposit(e, amountA + amountB) at init; // reset storage to init and deposit sum

  uint together_balance = balanceOf(e.msg.sender); // save resulting balance

  assert separate_balance == together_balance, // compare
    "splitting a deposit into two smaller deposits must have the same effect on user's balance";
}
```

TYPES OF RULES

- ▶ [Most concrete] “Unit test style” / method specification properties
 - ▶ Encode the method documentation
 - ▶ Consider successful cases and revert conditions
- ▶ Variable changes
 - ▶ For each variable / getter, ask when and how it should change
 - ▶ Describe quality of changes and methods that can change
- ▶ Variable relationships
 - ▶ Ask what variables are related to each other and how
- ▶ State-transition diagrams
 - ▶ Ask how a contract (or part of the state) can evolve
- ▶ Risk assessment
 - ▶ Take perspective of different stakeholders
- ▶ [Most abstract] Mathematical properties
 - ▶ Abstract away from details [e.g. monotonicity, additivity, commutativity]

HOMEWORK

For next time:

- ▶ Design rule set for SymbolicPool (checked in soon)