Quick Guide to the Certora Prover

This document explains how to use the Certora prover web IDE through a simple example called Bank.

Rules in Specify

The Certora prover checks that a smart contract satisfies a set of rules written in a language called Specify. Each rule is checked on all possible transactions, though of course this is not done by explicitly enumerating transactions, but rather through symbolic techniques. Rules can check that a public contract method has the correct effects on the contract state, or that it returns the correct value, etc. The syntax for expressing rules somewhat resembles Solidity, but also supports more features that are important for verification.

Consider the following contract interface to implement a simple bank.

```
contract Bank {
    mapping (address => uint) public funds;
    uint public totalFunds;

    //get the current fund of an account
    function getFunds(address account) public returns (uint)
    //get the total fund in the bank
    function getTotalFunds() public returns (uint)
    //transfer amount to account
    function deposit(uint amount) public payable
    //transfer amount to account
    function transfer(address account, uint amount) public
    //withdraw all amount
    function withdraw() returns (bool) public
}
```

And here is a rule for verifying that `withdraw` either reverts or returns `true`.

```
//file bank.spec
rule withdraw_succeeds {
   env e; // env represents the bytecode environment passed one every call
   // invoke function withdraw and assume that it does not revert
   bool success = sinvoke withdraw(e);  // e is passed as an additional arg
   assert success, “withdraw must succeed”; // verify that withdraw succeed
}
```

The rule calls `withdraw` with an arbitrary EVM environment (`e`) in an arbitrary initial state. It assumes that the function does not revert (in the command `sinvoke` where “s” stands for successful). The `assert` command checks that success is true on all potential executions. Notice that each Solidity function has an extra argument which is the EVM environment.

Predefined Types

- **`env` -** represents the environment of the EVM during execution. This type contains the following fields, for an instance `env e`:
    - `e.msg.address` - address of the contract begin verified, e.g., Bank
    - `e.msg.sender` - sender of the message
    - `e.msg.value` - number of wei sent with the message
    - `e.block.number` - current block number
    - `e.block.timestamp` - current time stamp
    - `e.tx.origin` - original message sender
- **`method`** - represents methods and their attributes. This type contains the following fields for instance m of type method:
    - `m.name` - the name of the method `m`
    - `m.selector` - the hashcode of the function
    - `m.isPure` - true when m is declared with the pure attribute
    - `m.isView` - true when m is declared with the view attribute
    - `m.numberOfArguments` - the number of arguments to method `m`.

Standard Identifiers

The Specify language includes the following standard identifiers:

- **`bool lastReverted`** - true when the last function call reverted, for example “did the transfer revert?”.

```
//file bank.spec
//a rule with two free variables: to - the address the transfer is passed
//and the amount to pass.
rule transfer_reverts() {
   env e;
   address to;
   uint256 amount;
   // invoke function transfer and assume - caller is w.msg.from
   uint256 balance = sinvoke getFunds(e, e.msg.sender);
   invoke transfer(e, to, amount);
   // check that transfer reverts if not enough funds
   assert balance < amount => lastReverted , "insufficient funds";
}
```

- **`string lastRevertedString`** - the last revert message
- **`address currentContract`** - the address of the currentContract that is checked, e.g. the address of Bank.
- **`state lastStorage`** - The current state of the contract. Useful for enforcing hyperproperties of smart contracts.

Specify commands

- **`require`** `exp` - assume that `exp` is true at this point (i.e., the tool will only consider executions in which `exp` holds). For example, require `e.msg.sender == admin` would ignore any cases where the caller is not the admin.
- **`assert`** `exp` - check if `exp` is true. For example, `assert newBalance == oldBalance + amount` will check that a balance always equals the correct value after a transfer (or will report an error, such as the case where an account transfers to itself and this assertion doesn't hold). The optional string argument is displayed when the assertion is violated.
- **`invoke`** `foo(args)` - process function `foo` with arguments `args`.
- **`sinvoke`** `foo(args)` - process function `foo` with arguments `args` and assume that it does not revert. `sinvoke` is actually equivalent to:

```
invoke foo(arg);
require !lastReverted;
```

Boolean operators beyond Solidity

- Implication: **=>**
  `A => B` evaluates to true if either `A` is false or `B` is true.

For example, `assert e.sender != admin => lastReverted` could check that if the caller is not the admin, a given function must revert in all cases.

- Bi-directional implication: **<=>**
  `A <=> B` evaluates to true iff `A => B && B => A`
  For example, `assert e.sender != admin <=> lastReverted` checks that if the caller is not the admin, a given function reverts **and** that if the function reverted, it must be the case that the sender was not the admin (basically saying that this is the only reason it would revert).

Parametric rules

In order to simulate the execution of all functions in the Contract, you can define a method argument in the rule and use it in `invoke` (`sinvoke`) statements. For example:

```
//file bank.spec
rule others_can_only_increase_balance() {
   method f; // an arbitrary function in the contract
   env e;  // the execution environment
   address other; // a different account
   // assume msg.sender is a different address
   require e.msg.sender != other;
   // get balance before
   uint256 _balance = sinvoke getFunds(e,other);
   // exec some method
   calldataarg arg; // any argument
   sinvoke f(e,arg); // successful (potentially state-changing!)
   //get balance after
   uint256 balance_ = sinvoke getFunds(e,other);
   //balance should not be reduced by any operation
   //may increase due to a transfer from msg.sender to other
   assert _balance <= balance_ ,"withdraw from others balance";
}
```

This is the equivalent of calling **any** function and the prover verifies that the conditions hold against **any function call with any arguments.**

Invariants in Specify

Invariants are a specification of a condition that should always be true once an operation is concluded. Syntax:

- `invariant invariantName(args_list) exp` - Assume `exp` holds before execution of any method and verify `exp` must hold afterwards
  The invariant above is equivalent to a rule:

```
method f;
require exp;
calldataarg arg;
sinvoke f(e,arg);
assert exp;
```

Declaring functions used in the invariant

In the spec file, you need to define the Solidity functions used in the invariants. Here `env` and `z` ranges over all environments and addresses.

```
//file bank.spec
methods {
      init_state()
      getFunds(address) returns uint256
}

invariant address_zero_cannot_become_an_account(env e, address z)
          z==0 => sinvoke getFunds(e, z)==0
```

envfree functions

When a function is not using the environment it can be declared as **`envfree`** and omit the `env` argument in the call. For example, `getFunds` is not using any of the environment variables:

```
//file bank.spec
methods {
      init_state()
      getFunds(address) returns uint256 envfree
}
invariant address_zero_cannot_become_an_account(address a)
          a==0 => sinvoke getFunds(a)==0
```

Understanding the results

The Certora prover produces a table with the verification results in an .html file. For each rule, it either displays a thumbs-up when it formally proved the rule or displays an input which triggers a violation of the rule. For example, below is a violation of rule `others_can_only_increase` when simulated on the `transfer` function. A call trace demonstrating the violation is shown. It shows the function simulated on which arguments and the return value (displayed after the slash). In this case it shows that a transfer to some address an amount that is close to `max_int` causes the balance of that account to decrease.


Results for others_can_only_increase:

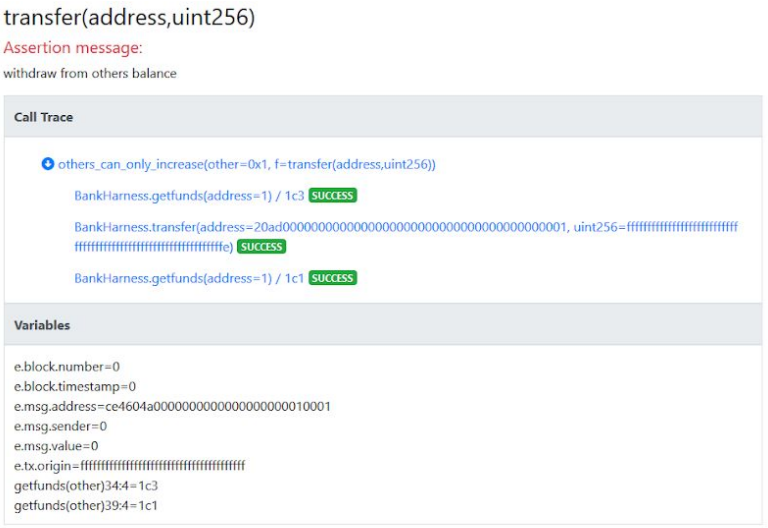
Go back to top

Function name	Result	Time(secs)	Dump
transfer(address,uint256)	👎	0	
withdraw()	👍	0	
_ercBalance()	👍	0	
totalFunds()	👍	0	
init_state()	👍	0	
deposit(uint256)	👍	0	
funds(address)	👍	0	
getTotalFunds()	👍	0	
getfunds(address)	👍	0	
certorafallback_0()	👍	0	

Results for can_withdraw_after_any_time_and_any_other_transaction:

Go back to top

Function name	Result	Time(secs)	Dump
transfer(address,uint256)	👎	1	
withdraw()	👎	1	
_ercBalance()	👎	0	


transfer(address,uint256)

Assertion message:

withdraw from others balance

Call Trace

others_can_only_increase(other=0x1, f=transfer(address,uint256))

BankHarness.getfunds(address=1) / 1c3 SUCCESS

BankHarness.transfer(address=20ad0000000000000000000000000000000000000001, uint256=fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe) SUCCESS

BankHarness.getfunds(address=1) / 1c1 SUCCESS

Variables

e.block.number=0
e.block.timestamp=0
e.msg.address=ce4604a00000000000000000010001
e.msg.sender=0
e.msg.value=0
e.tx.origin=ffffffffffffffffffffffffffffffffffffffff
getfunds(other)34:4=1c3
getfunds(other)39:4=1c1

Best Practices

When to use `env` argument

- The usage of `env` arguments allows to explicitly access EVM parameters such as `msg.sender`.
- `env` arguments also allow to describe the behavior of multiple EVM transactions. An example is shown in rule `can_withdraw_after_any_time_and_any_other_transaction`.

```
rule can_withdraw_after_any_time_and_any_other_transaction() {
       address account;
       uint256 amount;
       method f;

       // account deposits amount
       env _e;
       require _e.msg.sender == account;
       require amount > 0;
       sinvoke deposit(_e,amount);

       //any other transaction beside withdraw & transfer by account
       env eF;
       require (f.selector != withdraw().selector &&
                f.selector != transfer(address, uint256).selector)
               || eF.msg.sender!=account;

       calldataarg arg; // any argument
       sinvoke f(eF,arg); // successful (potentially state-changing!)

       //account withdraws
       env e_;
       require e_.block.timestamp > _e.block.timestamp ; // The operation occurred
after the initial operation
       require e_.msg.sender == account;
       sinvoke withdraw(e_);
       // check the erc balance
       uint256 ethBalance = sinvoke getEthBalance(e_);
       assert ethBalance >= amount, "should have at least what have been
deposited";
}
```

Hyperproperties

It is possible to compare the effects of different transactions starting on the same state. The `additiveTransfer` rule checks that the `transfer` command is additive.

```
rule additiveTransfer(uint256 amt1, uint256 amt2, address from, address to) {
    env e1;
    env e2;

    // e1 and e2 transfer from the same address `from`
    require e1.msg.sender == from && e2.msg.sender == from;

    // record state before the transaction
    storage init = lastStorage;

    // Transfer amt1 and then amt2 from `from` to `to`
    sinvoke transfer(e1,to,amt1);
    sinvoke transfer(e2,to,amt2);
    uint256 balanceToCase1 = sinvoke getFunds(to);
    uint256 balanceFromCase1 = sinvoke getFunds(from);

    // Start a new transaction from the initial state
    sinvoke transfer(e1, to, amt1+amt2) at init;
    uint256 balanceToCase2 = sinvoke getFunds(to);
    uint256 balanceFromCase2 = sinvoke getFunds(from);
    assert balanceToCase1 == balanceToCase2 &&
             balanceFromCase1 == balanceFromCase2,
             "expected transfer to be additive" ;
}
```