Verification Techniques for Smart Contracts

12.06.2021

—

Waqas Ahmed
Course # DLT5402
Student # 1911706

Introduction

The Escrow Smart Contract

Specifications:

- 2 Actors: Sender, Receiver;
- Only Sender can place money in contract;
- Only Receiver can withdraw from contract;
- Receiver can only withdraw after a specified delay from the event of money being placed;
- Contract cannot be used again after withdrawal;

The PiggyBank Smart Contract

Specifications:

- 1 Actor: Owner;
- Only Owner can add money in contract;
- Only Owner can withdraw from contract;
- Owner can only withdraw after a delay of 365 days from the event of money being deposited/added the first time;
- Contract cannot be used again after breaking piggy bank (withdrawal);

Note: The contract seem missing a few checks,

- addMoney() has a "require" missing of non-zero msg.value
- breakPiggyBank() is missing the setting of "balance" to zero.
- breakPiggyBank() is missing the resetting of "timeofFirstDeposit".
- breakPiggyBank() is missing the resetting of state to "Unused".

The last three probably don't matter since the piggyBank is made of clay, hence not reusable :) However, if it was a reusable moneyBank, these would be required.

- *Another major change in terms of best practice following EIP 1884 is to stop using .transfer() or .send() to transfer funds (which is used in both Escrow and PiggyBank). https://github.com/ConsenSys/smart-contract-best-practices/blob/master/docs/recommendations.md#dont-use-transfer-or-send*

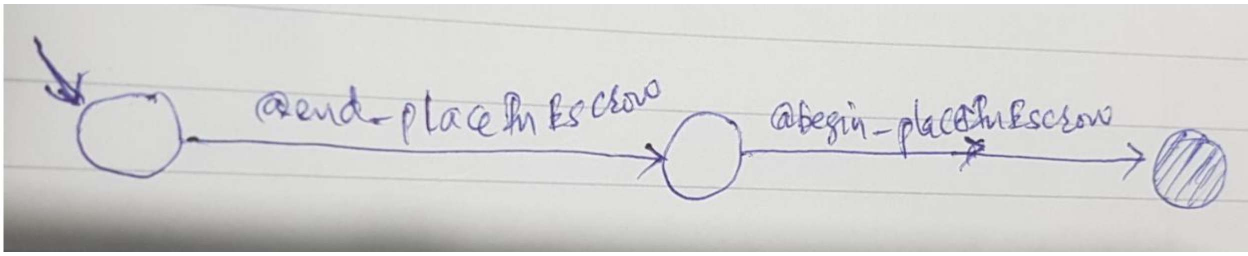
Code Repository for Tests: https://github.com/wakqasahmed/dlt_verification_techniques

Question 0: Specification Languages

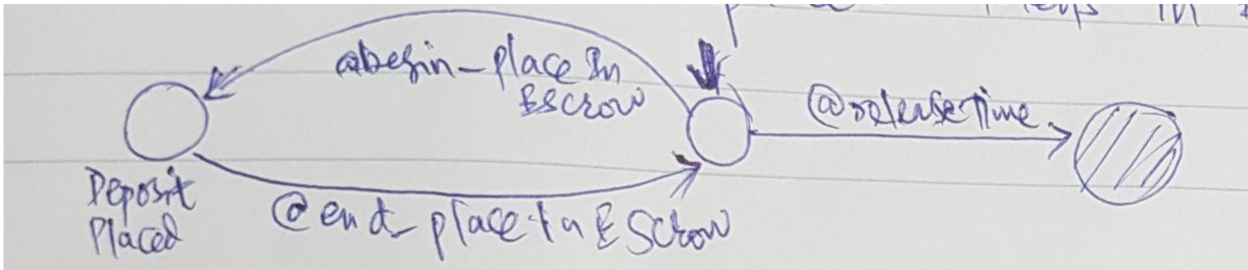
Properties for Escrow smart contract

I. Finite state automata

A. After a successful “placeInEscrow”, any further calls to “placeInEscrow” should fail.

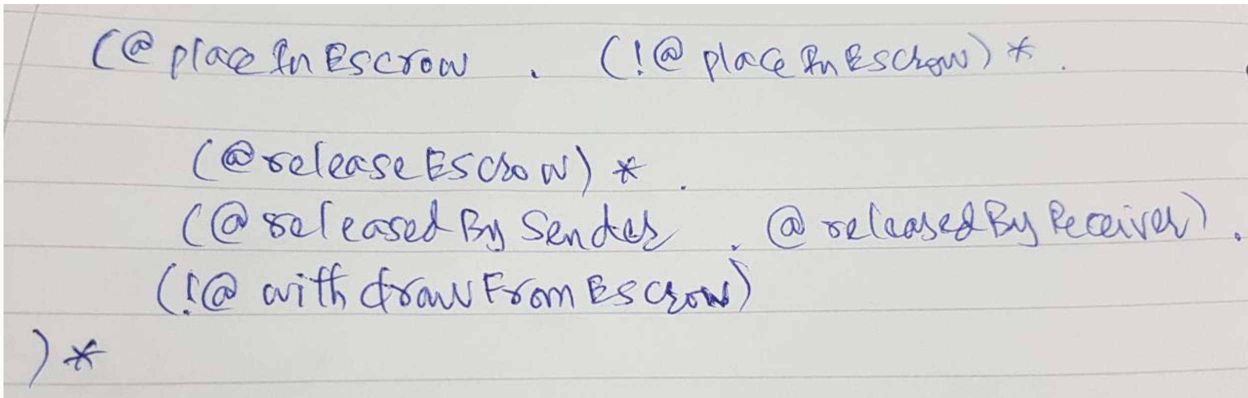


B. The timestamp of ReleaseTime may only be set when the “sender” places token(s) in Escrow i.e. the ReleaseTime must not change when the body of placeInEscrow function is not under execution, in case the releaseTime timestamp changes when we are still at initial state, something has gone wrong.



II. Regular expressions (Positive)

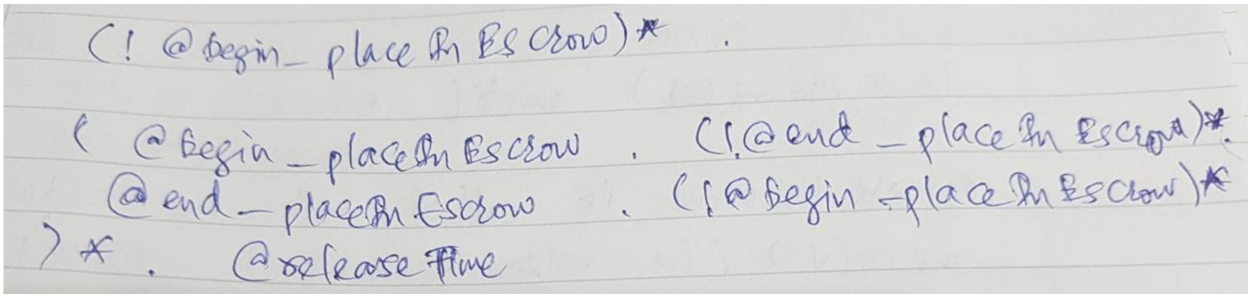
A. After the successful “placeInEscrow”, “withdrawFromEscrow” call should fail until “releaseEscrow” is called and “releaseBySender” & “releaseByReceiver” is set to true.



```
(@placeInEscrow . (!@placeInEscrow)* .
    (@releaseEscrow)* .
    (@releasedBySender . @releasedByReceiver) .
   (!@withdrawFromEscrow)
)*
```

III. Regular expressions (Negative)

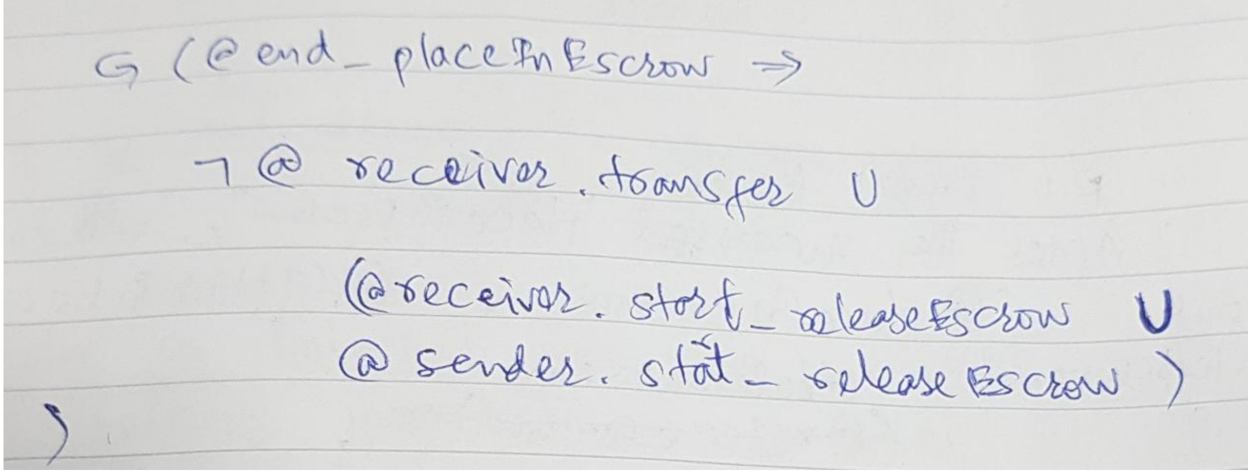
A. The releaseTime timestamp may only be set when the sender places token(s) in Escrow.



```
(! @begin_placeInEscrow)* .
( @begin_placeInEscrow . (!@end_placeInEscrow)* .
  @end_placeInEscrow . (!@begin_placeInEscrow)*
)* . @releaseTime
```

IV. LTL

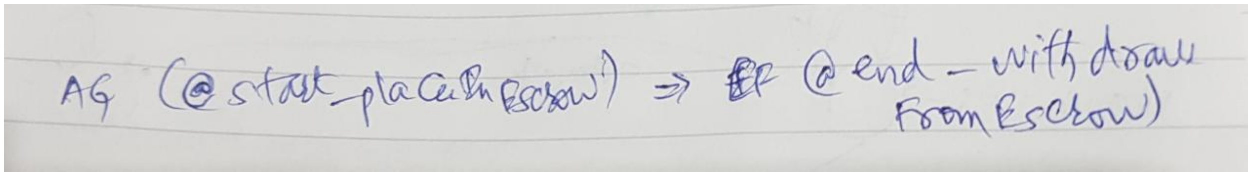
A. From any moment when “placeInEscrow” is called, there can be no outgoing funds (@receiver.transfer) until “releaseEscrow” is called by sender and receiver both (i.e. releasedBySender and releasedByReceiver is set to true)



```
G (@end_placeInEscrow =>
      ¬ @receiver.transfer U
           (@receiver.start_releaseEscrow U
           @sender.start_releaseEscrow )
)
```

V. CTL

A. Once the tokens are placed in Escrow, there is always a way for the receiver to withdraw funds.

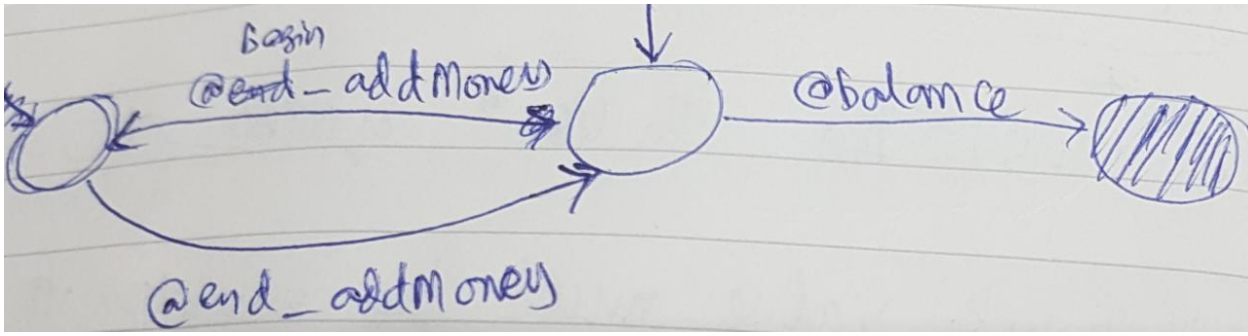


AG (@start_placeInEscrow) => EF @end_withdrawFromEscrow)

Properties of PiggyBank smart contract

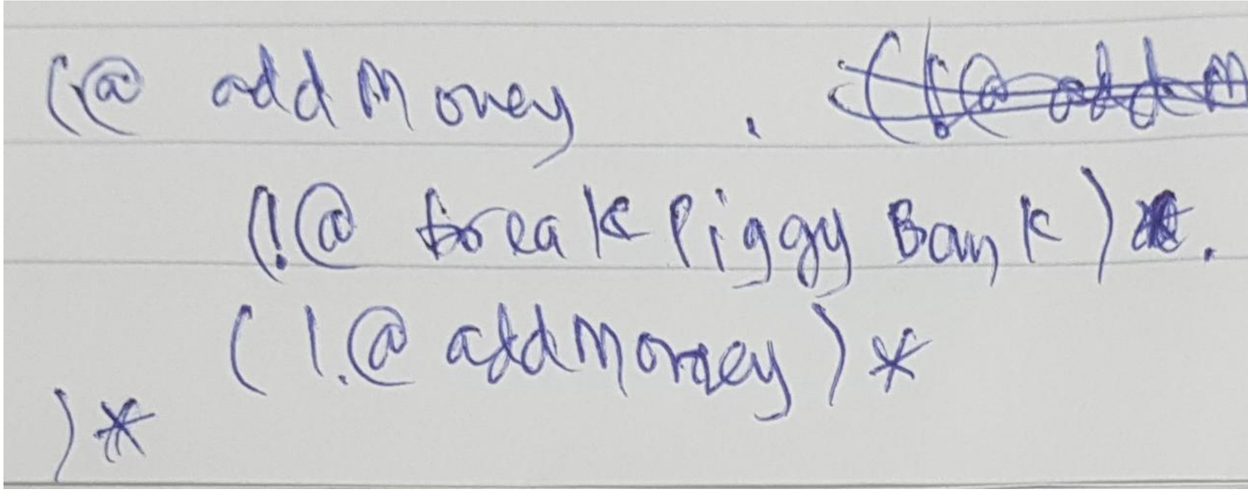
I. Finite state automata

A. The value of balance may only be changed when the owner adds tokens to Escrow i.e. the balance may only be changed when the addMoney body is being executed.



II. Regular expressions (Positive)

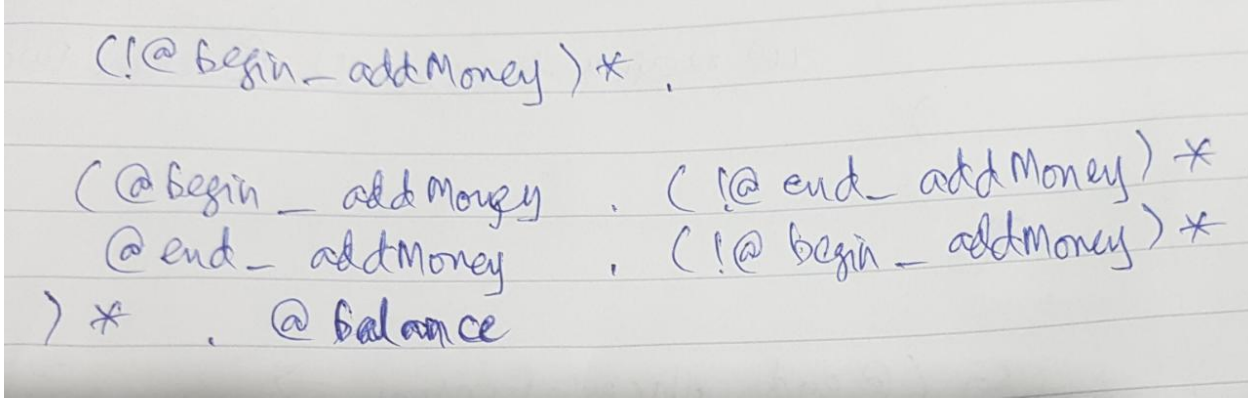
A. After the successful “addMoney” for the first time, “addMoney” call should fail if the “breakPiggyBank” function has already been called, else, the “owner” can continue to “addMoney” to the piggyBank.



(@addMoney . ~~(!@addM~~
(!@breakPiggyBank)*.
(!.@addMoney)*
)*

III. Regular expressions (Negative)

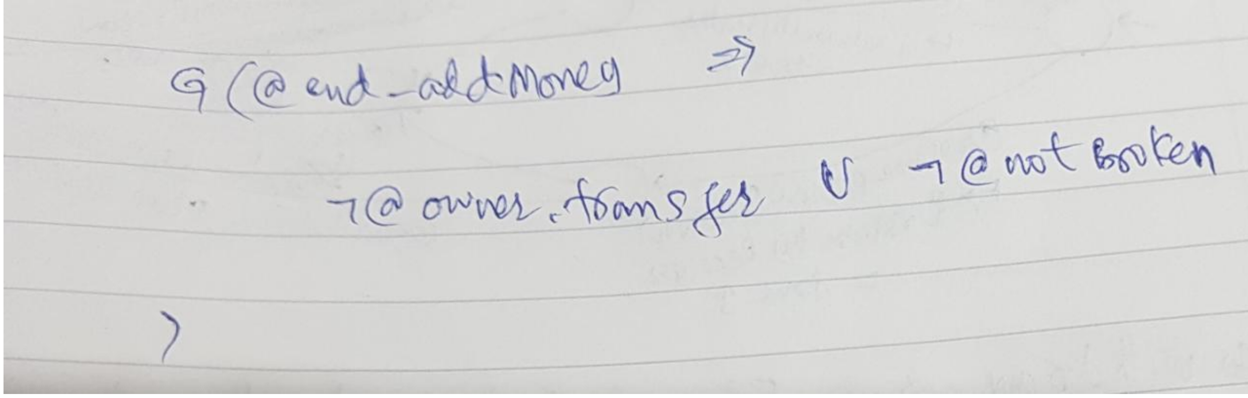
A. The balance may only be changed when the owner adds money in the piggyBank.



```
(!@begin_addMoney)* .
(@begin_addMoney . (!@end_addMoney)*
 @end_addMoney   . (!@begin_addMoney)*
)* . @balance
```

IV. LTL

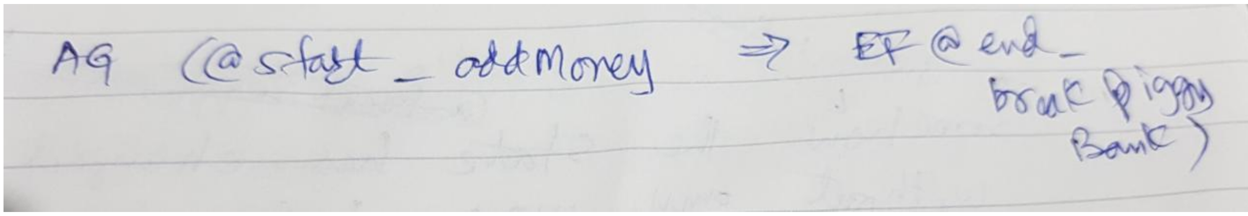
A. From any moment when “addMoney” is called, there can be no outgoing funds (@owner.transfer) until the state is set to “Broken”.



```
G (@end_addMoney  =>
      ¬@owner.transfer  U  ¬@not Broken
)
```

V. CTL

A. Once the money is added, there is always a way for the owner to breakPiggyBank.



```
AG (@start_addMoney  =>  EF @end_
                            breakPiggy
                              Bank)
```

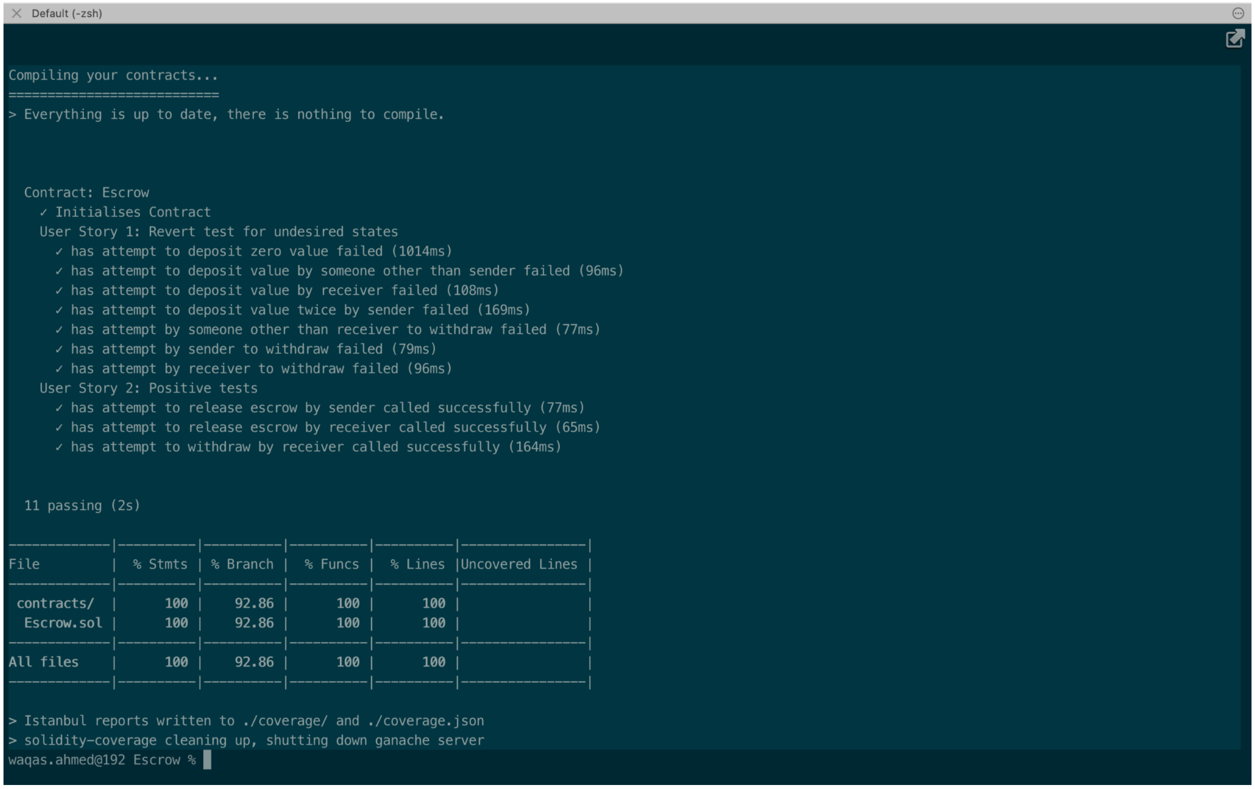
Question 1: Testing

Write tests, including ones based on specifications written as automata.

Tests for Escrow smart contract

Discuss

(i) the test specifications written;



```
Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.



  Contract: Escrow
    ✓ Initialises Contract
    User Story 1: Revert test for undesired states
      ✓ has attempt to deposit zero value failed (1014ms)
      ✓ has attempt to deposit value by someone other than sender failed (96ms)
      ✓ has attempt to deposit value by receiver failed (108ms)
      ✓ has attempt to deposit value twice by sender failed (169ms)
      ✓ has attempt by someone other than receiver to withdraw failed (77ms)
      ✓ has attempt by sender to withdraw failed (79ms)
      ✓ has attempt by receiver to withdraw failed (96ms)
    User Story 2: Positive tests
      ✓ has attempt to release escrow by sender called successfully (77ms)
      ✓ has attempt to release escrow by receiver called successfully (65ms)
      ✓ has attempt to withdraw by receiver called successfully (164ms)


  11 passing (2s)

-------------|----------|----------|----------|----------|----------------|
File         |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
-------------|----------|----------|----------|----------|----------------|
 contracts/  |      100 |    92.86 |      100 |      100 |                |
  Escrow.sol |      100 |    92.86 |      100 |      100 |                |
-------------|----------|----------|----------|----------|----------------|
All files    |      100 |    92.86 |      100 |      100 |                |
-------------|----------|----------|----------|----------|----------------|

> Istanbul reports written to ./coverage/ and ./coverage.json
> solidity-coverage cleaning up, shutting down ganache server
waqas.ahmed@192 Escrow % 
```

Test cases were divided into two stories:

1) Negative tests which should result in revert as they don't adhere to the desired functionality.
2) Positive tests which should execute successfully and match the intended result after execution.

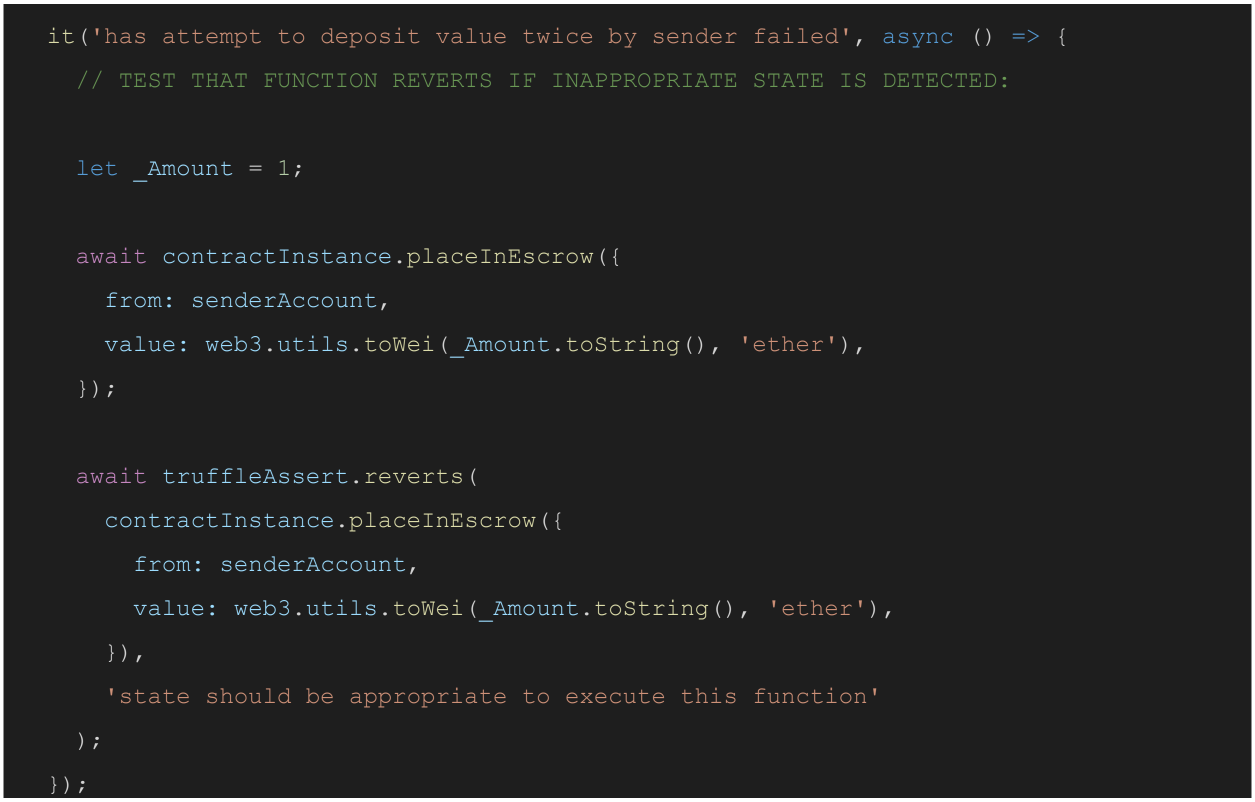
Please find the test cases on the below link:

https://github.com/wakqasahmed/dlt_verification_techniques/blob/main/testing/Escrow/test/Escrow.js

Moreover, there is a README file on how to execute the test cases and coverage test on the following link:

https://github.com/wakqasahmed/dlt_verification_techniques/blob/main/README.md

For instance, the following test case is written based on the finite state automata specification which ensures that the money cannot be deposited in escrow more than once.



```
it('has attempt to deposit value twice by sender failed', async () => {
  // TEST THAT FUNCTION REVERTS IF INAPPROPRIATE STATE IS DETECTED:

  let _Amount = 1;

  await contractInstance.placeInEscrow({
    from: senderAccount,
    value: web3.utils.toWei(_Amount.toString(), 'ether'),
  });

  await truffleAssert.reverts(
    contractInstance.placeInEscrow({
      from: senderAccount,
      value: web3.utils.toWei(_Amount.toString(), 'ether'),
    }),
    'state should be appropriate to execute this function'
  );
});
```

(ii) argue about the completeness of the specifications (i.e. adherence to the properties should imply that the system works correctly);

Contract: Escrow

✓ Initialises Contract

User Story 1: Revert test for undesired states

✓ has attempt to deposit zero value failed (1014ms)

✓ has attempt to deposit value by someone other than sender failed (96ms)

✓ has attempt to deposit value by receiver failed (108ms)

✓ has attempt to deposit value twice by sender failed (169ms)

✓ has attempt by someone other than receiver to withdraw failed (77ms)

✓ has attempt by sender to withdraw failed (79ms)

✓ has attempt by receiver to withdraw failed (96ms)

User Story 2: Positive tests

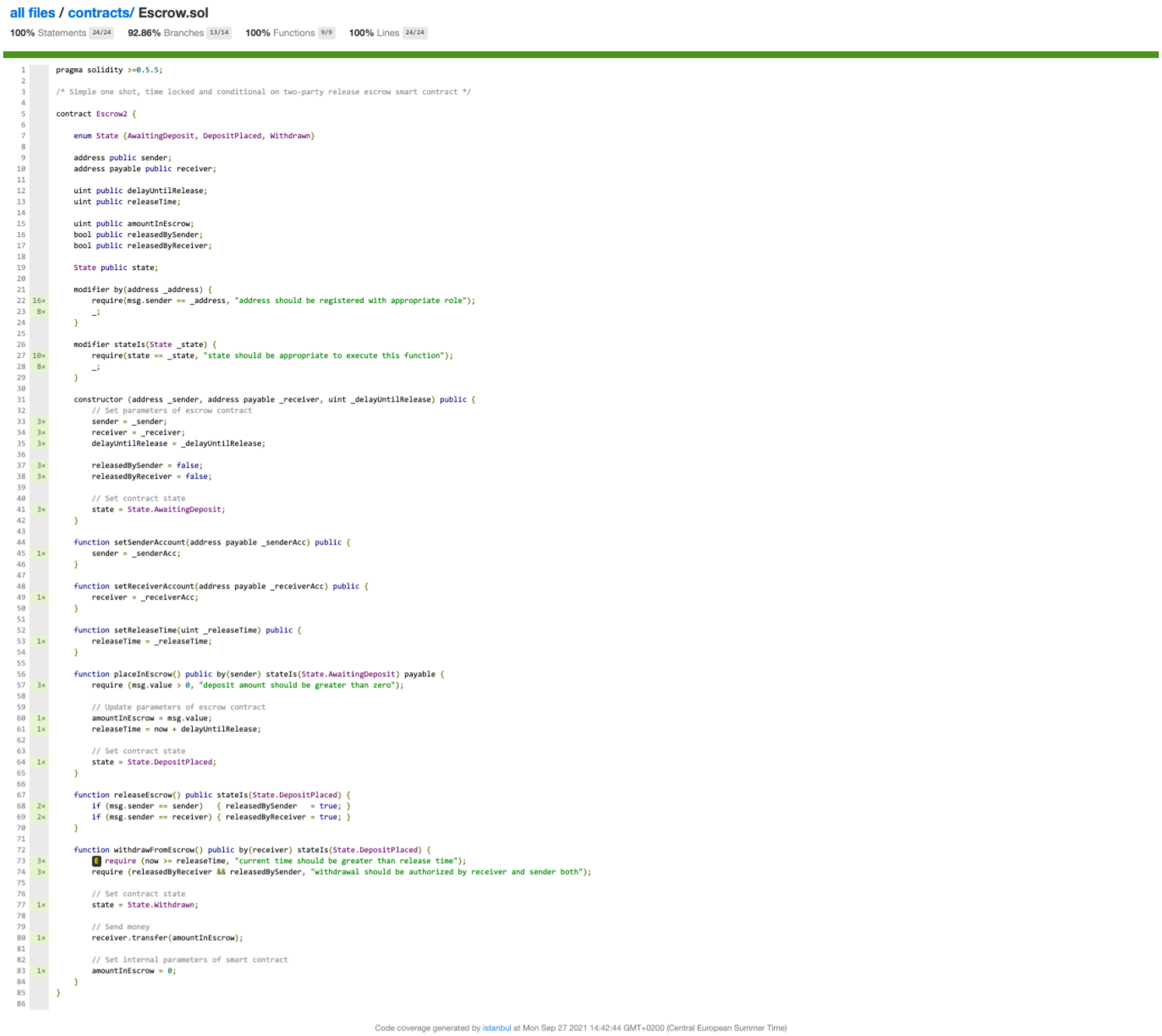
✓ has attempt to release escrow by sender called successfully (77ms)

✓ has attempt to release escrow by receiver called successfully (65ms)

✓ has attempt to withdraw by receiver called successfully (164ms)

(iii) coverage achieved by the tests. You may use coverage and security tools to support your reasoning.

Based on the coverage tool, 100% statements, 100% functions and 100% lines are covered, however, one branch (a require) is not covered as it involves future time which is a bit tricky to test.



all files / contracts/ Escrow.sol

100% Statements 24/24   92.86% Branches 13/14   100% Functions 9/9   100% Lines 24/24

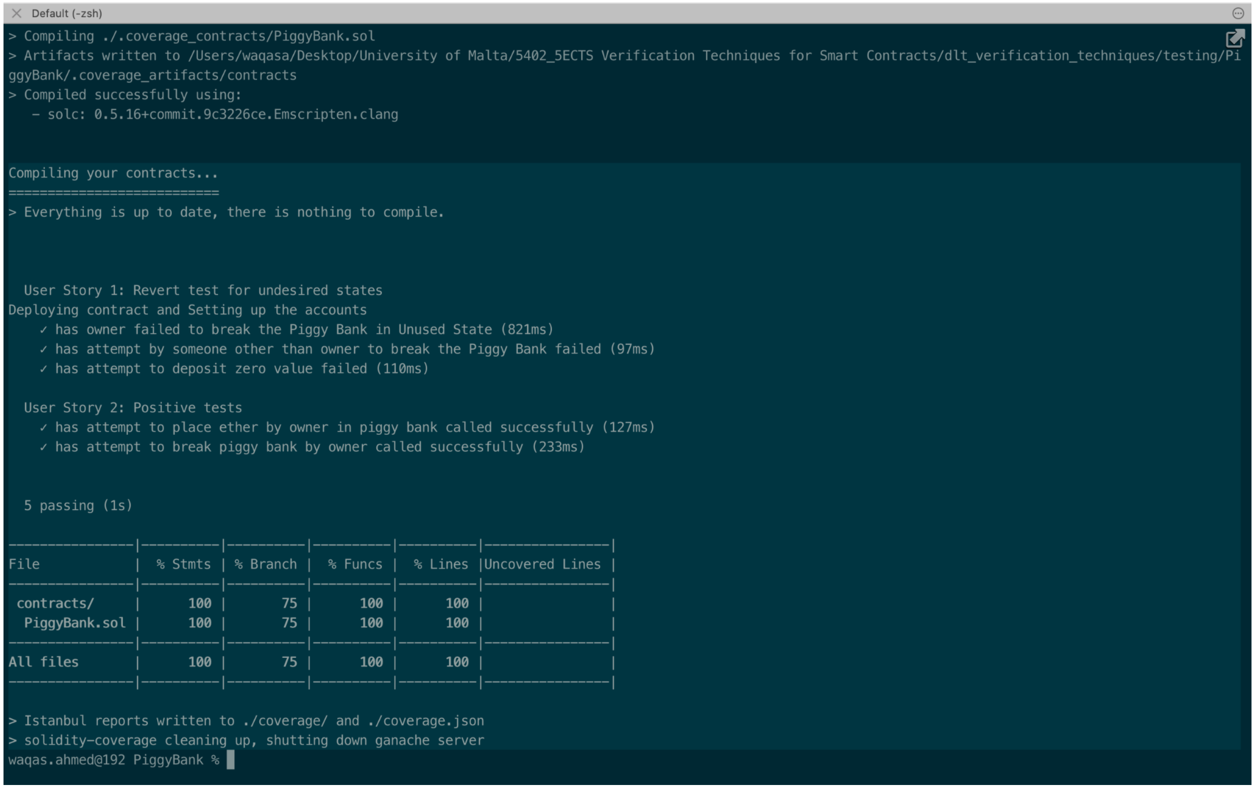
```
     pragma solidity >=0.5.5;

     /* Simple one shot, time locked and conditional on two-party release escrow smart contract */

     contract Escrow2 {

         enum State {AwaitingDeposit, DepositPlaced, Withdrawn}

         address public sender;
         address payable public receiver;

         uint public delayUntilRelease;
         uint public releaseTime;

         uint public amountInEscrow;
         bool public releasedBySender;
         bool public releasedByReceiver;

         State public state;

         modifier by(address _address) {
16×          require(msg.sender == _address, "address should be registered with appropriate role");
 8×          _;
         }

         modifier stateIs(State _state) {
10×          require(state == _state, "state should be appropriate to execute this function");
 8×          _;
         }

         constructor (address _sender, address payable _receiver, uint _delayUntilRelease) public {
             // Set parameters of escrow contract
 3×          sender = _sender;
 3×          receiver = _receiver;
 3×          delayUntilRelease = _delayUntilRelease;

 3×          releasedBySender = false;
 3×          releasedByReceiver = false;

             // Set contract state
 3×          state = State.AwaitingDeposit;
         }

         function setSenderAccount(address payable _senderAcc) public {
 1×          sender = _senderAcc;
         }

         function setReceiverAccount(address payable _receiverAcc) public {
 1×          receiver = _receiverAcc;
         }

         function setReleaseTime(uint _releaseTime) public {
 1×          releaseTime = _releaseTime;
         }

         function placeInEscrow() public by(sender) stateIs(State.AwaitingDeposit) payable {
 3×          require (msg.value > 0, "deposit amount should be greater than zero");

             // Update parameters of escrow contract
 1×          amountInEscrow = msg.value;
 1×          releaseTime = now + delayUntilRelease;

             // Set contract state
 1×          state = State.DepositPlaced;
         }

         function releaseEscrow() public stateIs(State.DepositPlaced) {
 2×          if (msg.sender == sender)   { releasedBySender   = true; }
 2×          if (msg.sender == receiver) { releasedByReceiver = true; }
         }

         function withdrawFromEscrow() public by(receiver) stateIs(State.DepositPlaced) {
 3×          E require (now >= releaseTime, "current time should be greater than release time");
 3×          require (releasedByReceiver && releasedBySender, "withdrawal should be authorized by receiver and sender both");

             // Set contract state
 1×          state = State.Withdrawn;

             // Send money
 1×          receiver.transfer(amountInEscrow);

             // Set internal parameters of smart contract
 1×          amountInEscrow = 0;
         }
     }
```

Code coverage generated by istanbul at Mon Sep 27 2021 14:42:44 GMT+0200 (Central European Summer Time)

Tests for PiggyBank smart contract

Discuss

(i) the test specifications written;



```
> Compiling ./.coverage_contracts/PiggyBank.sol
> Artifacts written to /Users/waqasa/Desktop/University of Malta/5402_5ECTS Verification Techniques for Smart Contracts/dlt_verification_techniques/testing/Pi
ggyBank/.coverage_artifacts/contracts
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang


Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.



  User Story 1: Revert test for undesired states
Deploying contract and Setting up the accounts
    ✓ has owner failed to break the Piggy Bank in Unused State (821ms)
    ✓ has attempt by someone other than owner to break the Piggy Bank failed (97ms)
    ✓ has attempt to deposit zero value failed (110ms)

  User Story 2: Positive tests
    ✓ has attempt to place ether by owner in piggy bank called successfully (127ms)
    ✓ has attempt to break piggy bank by owner called successfully (233ms)


  5 passing (1s)

----------------|----------|----------|----------|----------|----------------|
File            |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------|----------|----------|----------|----------|----------------|
 contracts/     |      100 |       75 |      100 |      100 |                |
  PiggyBank.sol |      100 |       75 |      100 |      100 |                |
----------------|----------|----------|----------|----------|----------------|
All files       |      100 |       75 |      100 |      100 |                |
----------------|----------|----------|----------|----------|----------------|

> Istanbul reports written to ./coverage/ and ./coverage.json
> solidity-coverage cleaning up, shutting down ganache server
waqas.ahmed@192 PiggyBank % █
```

Test cases were divided into two stories:

1) Negative tests which should result in revert as they don't adhere to the desired functionality.
2) Positive tests which should execute successfully and match the intended result after execution.

Please find the test cases on the below link:

https://github.com/wakqasahmed/dlt_verification_techniques/blob/main/testing/PiggyBank/test/PiggyBank.js

(ii) argue about the completeness of the specifications (i.e. adherence to the properties should imply that the system works correctly);

User Story 1: Revert test for undesired states

Deploying contract and Setting up the accounts

✓ has owner failed to break the Piggy Bank in Unused State (821ms)

✓ has attempt by someone other than owner to break the Piggy Bank failed (97ms)

✓ has attempt to deposit zero value failed (110ms)

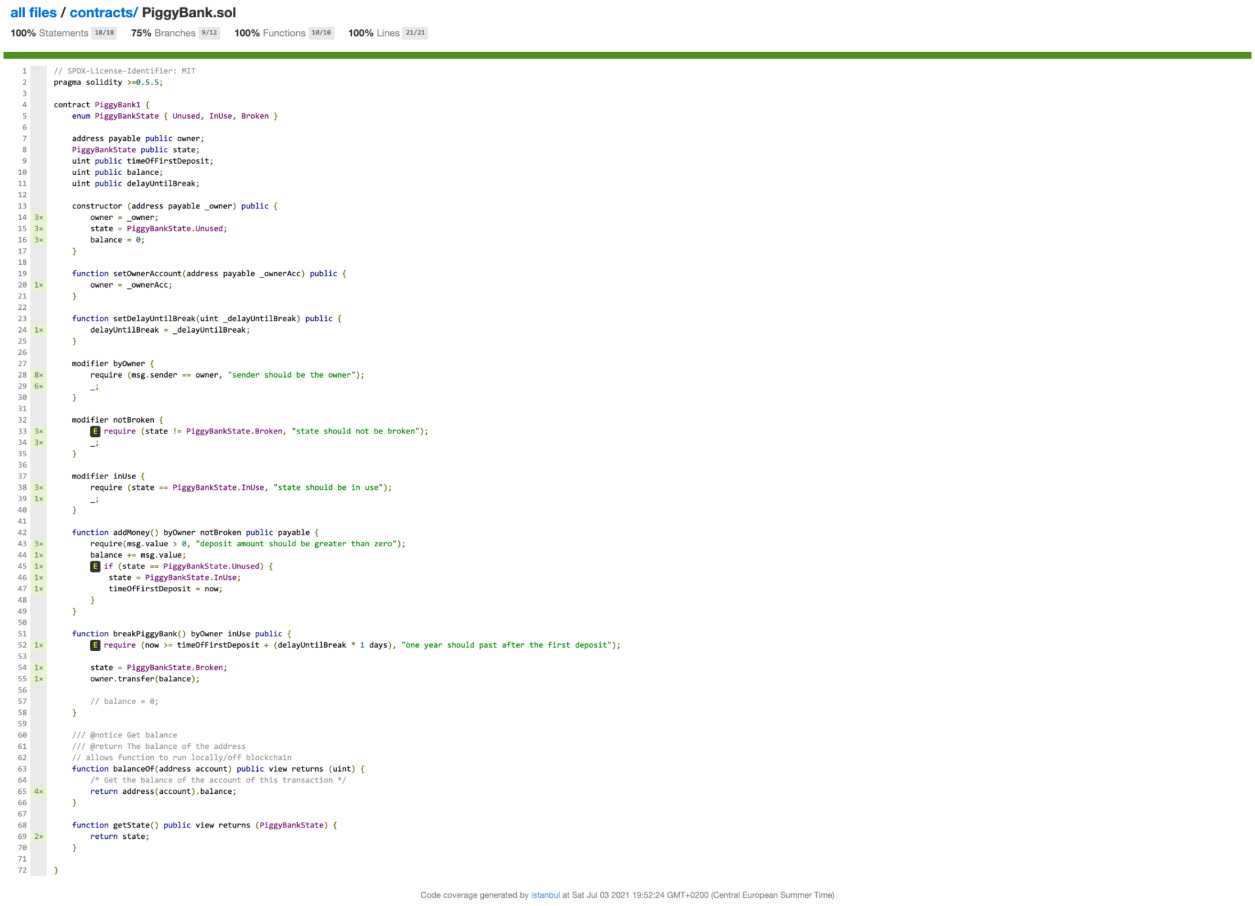
User Story 2: Positive tests

✓ has attempt to place ether by owner in piggy bank called successfully (127ms)

✓ has attempt to break piggy bank by owner called successfully (233ms)

(iii) coverage achieved by the tests. You may use coverage and security tools to support your reasoning.

Based on the coverage tool, 100% statements, 100% functions and 100% lines are covered, however, three branches are not covered.



all files / contracts/ PiggyBank.sol

100% Statements 18/18 75% Branches 9/12 100% Functions 10/10 100% Lines 21/21

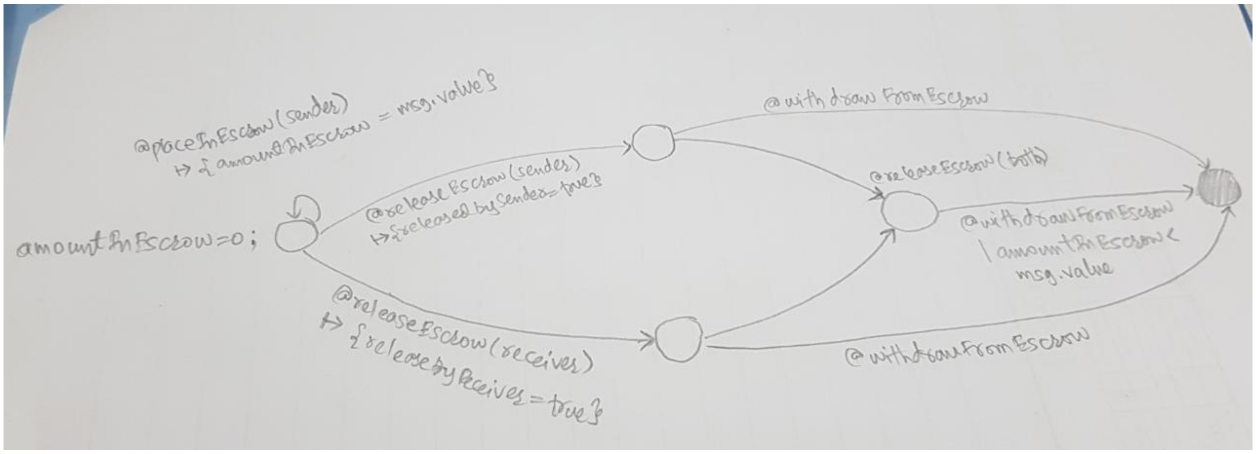
```
    // SPDX-License-Identifier: MIT
    pragma solidity >=0.5.5;

    contract PiggyBank1 {
        enum PiggyBankState { Unused, InUse, Broken }

        address payable public owner;
        PiggyBankState public state;
        uint public timeOfFirstDeposit;
        uint public balance;
        uint public delayUntilBreak;

        constructor (address payable _owner) public {
3×          owner = _owner;
3×          state = PiggyBankState.Unused;
3×          balance = 0;
        }

        function setOwnerAccount(address payable _ownerAcc) public {
1×          owner = _ownerAcc;
        }

        function setDelayUntilBreak(uint _delayUntilBreak) public {
1×          delayUntilBreak = _delayUntilBreak;
        }

        modifier byOwner {
8×          require (msg.sender == owner, "sender should be the owner");
6×          _;
        }

        modifier notBroken {
3×          E require (state != PiggyBankState.Broken, "state should not be broken");
3×          _;
        }

        modifier inUse {
3×          require (state == PiggyBankState.InUse, "state should be in use");
1×          _;
        }

        function addMoney() byOwner notBroken public payable {
3×          require(msg.value > 0, "deposit amount should be greater than zero");
1×          balance += msg.value;
1×          E if (state == PiggyBankState.Unused) {
1×              state = PiggyBankState.InUse;
1×              timeOfFirstDeposit = now;
            }
        }

        function breakPiggyBank() byOwner inUse public {
1×          E require (now >= timeOfFirstDeposit + (delayUntilBreak * 1 days), "one year should past after the first deposit");

1×          state = PiggyBankState.Broken;
1×          owner.transfer(balance);

            // balance = 0;
        }

        /// @notice Get balance
        /// @return The balance of the address
        // allows function to run locally/off blockchain
        function balanceOf(address account) public view returns (uint) {
            /* Get the balance of the account of this transaction */
4×          return address(account).balance;
        }

        function getState() public view returns (PiggyBankState) {
2×          return state;
        }

    }
```

Code coverage generated by istanbul at Sat Jul 03 2021 19:52:24 GMT+0200 (Central European Summer Time)

Question 2: Runtime Verification

Specifications for Escrow smart contract

Write specifications using Dynamic Event Automata.



1) While the state is DepositPlaced, amountInEscrow can not change in value.
2) The state remains DepositPlaced until Withdrawn.
3) Withdraw from Escrow is not allowed without releaseEscrow being called, both from sender and receiver, else we have reached a bad state.
4) amountInEscrow should not be less than msg.value at any point after Escrow is placed and before withdrawFromEscrow is executed, else we have reached a bad state.

Implementation for Escrow smart contract

Implement them using Contract Larva, adding reparations as you may deem appropriate.

Discuss

(i) the specifications written;

Code file link:
https://github.com/wakqasahmed/dlt_verification_techniques/blob/main/runtime_verification/Escrow/EscrowSpec.dea

```
monitor
Escrow2 {
    declarations {
        uint placedInEscrowTime;
        uint minimumReleaseDelayTime = 24*15 hours;

        address payable private sender_address;
        address payable private receiver;
        bool public releasedBySender;
        bool public releasedByReceiver;
        function getEscrowAmount() private returns(uint) { return value; }
        function getSender() private returns(address payable) { return
sender_address; }
        function getReceiver() private returns(address payable) { return
receiver; }

        enum STATE { AwaitingDeposit, DepositPlaced, Withdrawn }
        STATE private state = STATE.AwaitingDeposit;

        function placeInEscrow() payable public{
            require (state == STATE.AwaitingDeposit);
            require (msg.value == getEscrowAmount());
            require (msg.sender == getSender());
            state = STATE.DepositPlaced;
            LARVA_EnableContract();
        }
```

```
}


initialisation {
    sender_address = msg.sender;
}


reparation {
    getReceiver().transfer(getEscrowAmount());
    LARVA_DisableContract();
}


satisfaction {
    getReceiver().transfer(getEscrowAmount());
}


DEA Withdrawn{
    states{
        Start: initial;
        DepositPlaced;
        Withdrawn: accept;
        WithdrawalFailure: bad;
    }


    transitions{
            Start -[after(placed) | ~> placedInEscrowTime = now;]->
Deposited;

            Deposited -[after(withdrawal) | now - placedInEscrowTime <=
minimumReleaseDelayTime]-> Withdrawn;

            Deposited -[after(withdrawal) | now - placedInEscrowTime >=
minimumReleaseDelayTime]-> WithdrawalFailure;

        }

    }

}
```

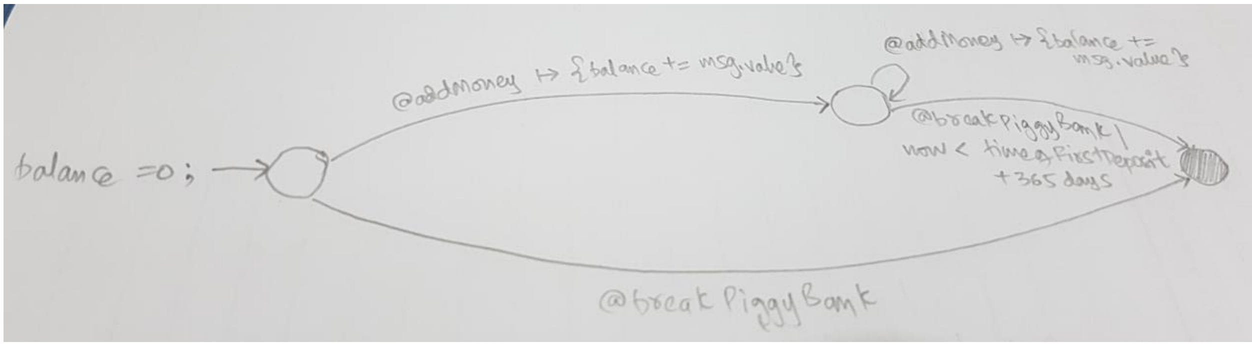
(ii) the reparations you implemented, and justify your choice; and

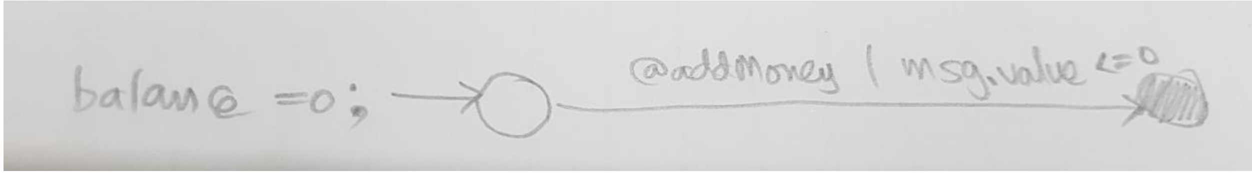
(iii) overheads induced.

Reparation was to transfer the amount to the receiver without further ado in case of withdrawal failure despite all conditions being met.

Specifications for PiggyBank smart contract

Write specifications using Dynamic Event Automata.


balance =0;
@addMoney ↦ {balance += msg.value}
@addMoney ↦ {balance += msg.value}
@breakPiggyBank | now < time@FirstDeposit +365 days
@break PiggyBank



1) While the state is InUse, balance can not reduce in value.

2) The state remains InUse until Broken.
3) breakPiggyBank must not get executed without the change of state to InUse, if it happens, we have reached a bad state i.e. in the state of Unused, piggyBank is unbreakable.
4) breakPiggyBank must not get executed if one year time has not elapsed, else bad state is reached.
5) addMoney function must not be called without a positive payable value (modifier is missing in the contract given with this assignment)

Implementation for PiggyBank smart contract

Implement them using Contract Larva, adding reparations as you may deem appropriate.

Discuss

(i) the specifications written;

Code file link:
https://github.com/wakqasahmed/dlt_verification_techniques/blob/main/runtime_verification/PiggyBank/PiggyBankSpec.dea

```
monitor
PiggyBank{



                  declarations{

                      uint total;

                      address payable private owner_address;

                      function getTotal() private returns(uint) { return value; }

                      function getOwner() private returns(address payable) { return
               owner_address; }

                  }



                  initialisation{

                      LARVA_EnableContract();
```

```
    }


    reparation {
        getOwner().transfer(getTotal());
        LARVA_DisableContract();
    }


    satisfaction {
        getOwner().transfer(getTotal());
    }


    DEA NoReduction {
        states {
            InUse: initial;
            Broken: accept;
            TimeOfFirstDeposit;
            BalanceReduced: bad;
        }
        transitions {
            InUse -[after(breakPiggyBank) | balance == 0 ]-> Broken;
            InUse -[after(addMoney(_value)) | _value <= balance ~> total +=
_value;]-> TimeOfFirstDeposit;
            TimeOfFirstDeposit -[after(addMoney(_value)) | _value <= balance
~> total += _value;]-> InUse;
```

```
                TimeOfFirstDeposit -[balance@(LARVA_previous_balance >
balance)]-> BalanceReduced;
            }
        }

        DEA InUseUntilBroken {
            states {
                Unused: initial;
                InUse: accept;
                TimeOfFirstDeposit;
                BrokenUnused: bad;
            }
            transitions {
                Unused -[after(addMoney)]-> InUse;
                Unused -[after(addMoney)]-> TimeOfFirstDeposit;
                Unused -[after(breakPiggyBank)]-> BrokenUnused;
            }
        }
}
```

(ii) the reparations you implemented, and justify your choice; and

(iii) overheads induced.

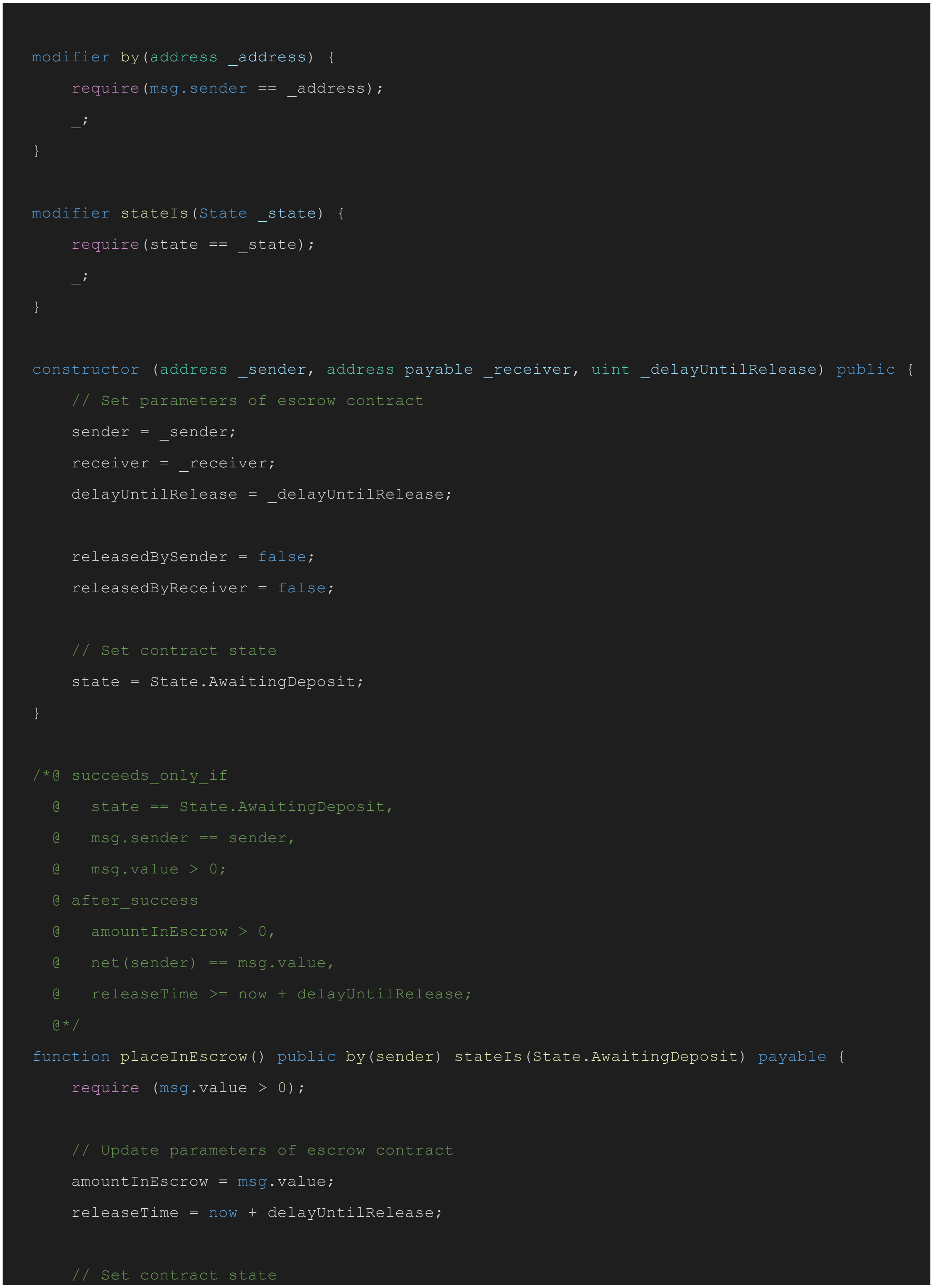
Reparation was to transfer the amount to the owner without further ado in case of reduction in amount in PiggyBank despite all conditions being met.

Question 3: Static Verification

Write functional specifications using the same notation and in a similar style as done for the Auction smart contract.

Functional Specifications for Escrow smart contract

```
pragma solidity >=0.5.5;

/* Simple one shot, time locked and conditional on two-party release escrow smart contract */
contract Escrow2 {

   enum State {AwaitingDeposit, DepositPlaced, Withdrawn}

   address public sender;
   address payable public receiver;

   uint public delayUntilRelease;
   uint public releaseTime;

   uint public amountInEscrow;
   bool public releasedBySender;
   bool public releasedByReceiver;

   State public state;

   /*@ invariant
     @   amountInEscrow == address(this).balance,
     @   sender == _sender,
     @   receiver == _receiver,
     @   delayUntilRelease == _delayUntilRelease;
     @*/

   /*@ invariant
     @   (\forall address a;
     @                 (a != sender && a != receiver && a != address(this))
     @             ==> net(a) == 0),
     @   state == AwaitingDeposit || state == Withdrawn ==> amountInEscrow == 0,
     @   state == DepositPlaced ==> amountInEscrow > 0;
     @*/
```



```
modifier by(address _address) {
    require(msg.sender == _address);
    _;
}

modifier stateIs(State _state) {
    require(state == _state);
    _;
}

constructor (address _sender, address payable _receiver, uint _delayUntilRelease) public {
    // Set parameters of escrow contract
    sender = _sender;
    receiver = _receiver;
    delayUntilRelease = _delayUntilRelease;

    releasedBySender = false;
    releasedByReceiver = false;

    // Set contract state
    state = State.AwaitingDeposit;
}

/*@ succeeds_only_if
  @   state == State.AwaitingDeposit,
  @   msg.sender == sender,
  @   msg.value > 0;
  @ after_success
  @   amountInEscrow > 0,
  @   net(sender) == msg.value,
  @   releaseTime >= now + delayUntilRelease;
  @*/
function placeInEscrow() public by(sender) stateIs(State.AwaitingDeposit) payable {
    require (msg.value > 0);

    // Update parameters of escrow contract
    amountInEscrow = msg.value;
    releaseTime = now + delayUntilRelease;

    // Set contract state
```

```
        state = State.DepositPlaced;
    }

    function releaseEscrow() public stateIs(State.DepositPlaced) {
        if (msg.sender == sender)   { releasedBySender   = true; }
        if (msg.sender == receiver) { releasedByReceiver = true; }
    }

    /*@ succeeds_only_if
      @   msg.sender == receiver,
      @   state == State.DepositPlaced,
      @   now >= releaseTime,
      @   releasedBySender == true || releasedByReceiver == true;
      @ after_success
      @   state == State.Withdrawn,
      @   net(receiver) == -net(sender),
      @   amountInEscrow == 0;
      @*/
    function withdrawFromEscrow() public by(receiver) stateIs(State.DepositPlaced) {
        require (now >= releaseTime);
        require (releasedByReceiver && releasedBySender);

        // Set contract state
        state = State.Withdrawn;

        // Send money
        receiver.transfer(amountInEscrow);

        // Set internal parameters of smart contract
        amountInEscrow = 0;
    }
}
```

Broken invariant in **withdrawFromEscrow()** function

```
      function withdrawFromEscrow() public by(receiver) stateIs(State.DepositPlaced) {
        require (now >= releaseTime);
        require (releasedByReceiver && releasedBySender);

        // Set contract state
        state = State.Withdrawn;
```

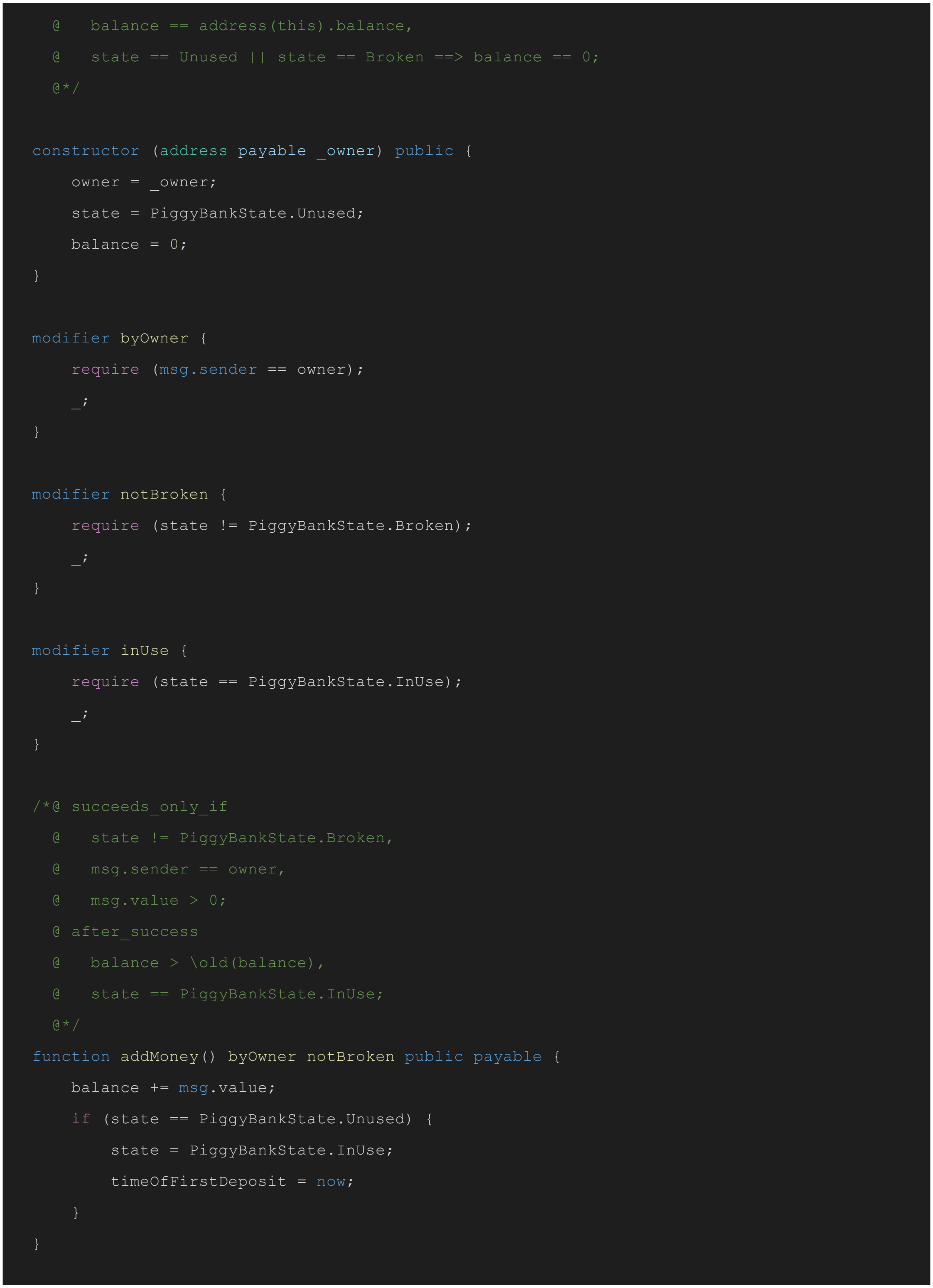
```
       // Send money
       receiver.transfer(amountInEscrow); // Broken invariant

       // Set internal parameters of smart contract
       amountInEscrow = 0;
   }
```

Fixed **withdrawFromEscrow()** function

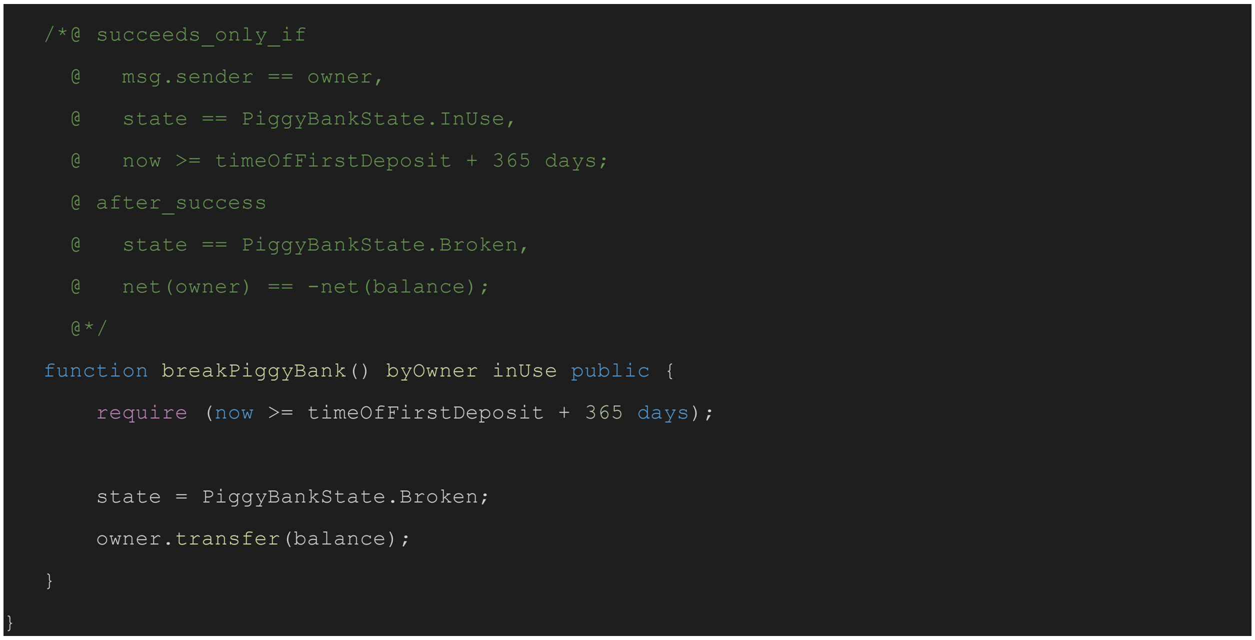
```
     function withdrawFromEscrow() public by(receiver) stateIs(State.DepositPlaced) {
       require (now >= releaseTime);
       require (releasedByReceiver && releasedBySender);

       // Set contract state
       state = State.Withdrawn;

       uint tmp = amountInEscrow;
       // Set internal parameters of smart contract
       amountInEscrow = 0;
       // Send money
       receiver.transfer(tmp);
   }
```

Functional Specifications for PiggyBank smart contract

```
pragma solidity >=0.5.5;

contract PiggyBank1 {
   enum PiggyBankState { Unused, InUse, Broken }

   address payable public owner;
   PiggyBankState public state;
   uint public timeOfFirstDeposit;
   uint public balance;

   /*@ invariant
     @   address(this) == owner;
     @*/

   /*@ invariant
```

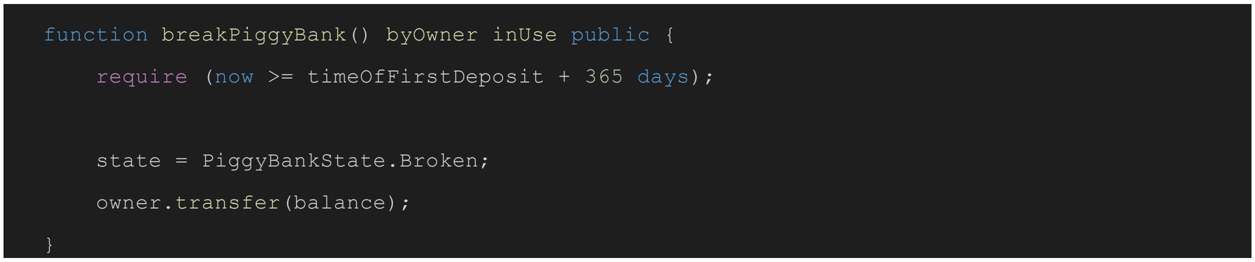


```
  @   balance == address(this).balance,
  @   state == Unused || state == Broken ==> balance == 0;
  @*/

constructor (address payable _owner) public {
    owner = _owner;
    state = PiggyBankState.Unused;
    balance = 0;
}

modifier byOwner {
    require (msg.sender == owner);
    _;
}

modifier notBroken {
    require (state != PiggyBankState.Broken);
    _;
}

modifier inUse {
    require (state == PiggyBankState.InUse);
    _;
}

/*@ succeeds_only_if
  @   state != PiggyBankState.Broken,
  @   msg.sender == owner,
  @   msg.value > 0;
  @ after_success
  @   balance > \old(balance),
  @   state == PiggyBankState.InUse;
  @*/
function addMoney() byOwner notBroken public payable {
    balance += msg.value;
    if (state == PiggyBankState.Unused) {
        state = PiggyBankState.InUse;
        timeOfFirstDeposit = now;
    }
}
```



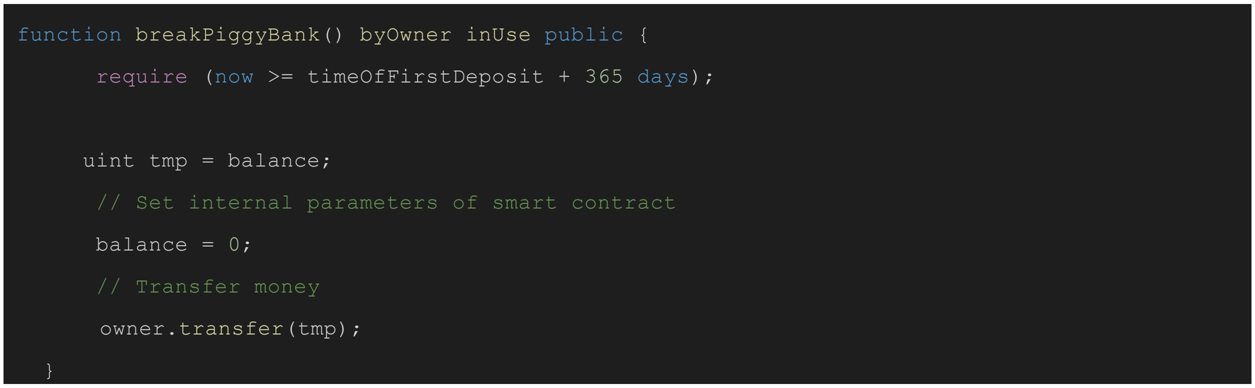
```
   /*@ succeeds_only_if
     @   msg.sender == owner,
     @   state == PiggyBankState.InUse,
     @   now >= timeOfFirstDeposit + 365 days;
     @ after_success
     @   state == PiggyBankState.Broken,
     @   net(owner) == -net(balance);
     @*/
   function breakPiggyBank() byOwner inUse public {
       require (now >= timeOfFirstDeposit + 365 days);

       state = PiggyBankState.Broken;
       owner.transfer(balance);
   }
}
```

Broken invariant in **breakPiggyBank()** function



```
   function breakPiggyBank() byOwner inUse public {
       require (now >= timeOfFirstDeposit + 365 days);

       state = PiggyBankState.Broken;
       owner.transfer(balance);
   }
```

Fixed **breakPiggyBank()** function



```
 function breakPiggyBank() byOwner inUse public {
       require (now >= timeOfFirstDeposit + 365 days);

      uint tmp = balance;
       // Set internal parameters of smart contract
       balance = 0;
       // Transfer money
       owner.transfer(tmp);
   }
```