contractLarva v0.2α
Tutorial

Shaun Azzopardi
shaun.azzopardi@um.edu.mt

19 November 2019, modified 24 October 2021

Abstract

This document illustrates contractLarva through several use-cases.

Contents

1 Overview

Smart contracts are programs, and like all programs they have a propensity for bugs and unintended behaviour, motivating the need for methods to detect and handle such behaviour. contractLarva is one such approach that uses runtime monitors to detect and deal with these bugs. Moreover, contractLarva can be used to adapt the behaviour of a smart contract, allowing a developer to develop smart contracts, or at least parts of a smart contract, using an event- and automata-based specification language. In this document we instead discuss several use cases, to illustrate the use and power of contractLarva.

This document is not intended to be an introduction to contractLarva and its syntax, which we assume the reader is familiar with. For that purpose read the contractLarva documentation at `https://www.github.com/gordonpace/contractlarva/docs/main.pdf`.

1.1 Background

Here we discuss and motivate several general areas of applicability contractLarva, including runtime verification, enforcement, and behavioural model synthesis. The use cases considered fall to different degrees under these areas.

The primary motivation for contractLarva is **runtime verification**. Verification is an approach to ensure well-behaved programs, by checking that the program is compliant with a specification. contractLarva is a tool for runtime verification of smart contracts, allowing behaviour recorded on the blockchain, relative to one smart contract, to be given a verdict (satisfying or violating) at runtime.

Ideally smart contracts are verified correct before they are deployed to the blockchain. However, this may not always be possible, either because the problem is too hard or because it involves interaction with other systems. Consider that a smart contract may be used to record real-life events, e.g. a courier service may allow a delivery person to signal delivery though an appropriate smart contract function. A specification can specify bad traces of these real-life events, e.g. we may specify that an item should not be delivered if it was not ordered. Such a smart contract can easily be designed to be compliant with this specification, however then it is no longer in sync with the real-world if a violation occurs, e.g. if the smart contract simply does not allow a delivery to be recorded (i.e. `revert` is called) without a prior order then it may not maintain correct stock records. This may affect compliance with other aspects of the desired behaviour of the smart contract.

Then, a specification may not necessarily simply be about a smart contract on its own, but also about its interaction with an outside system (e.g. the real-world). For these kinds of specifications methods based on **static code analysis should fail** (i.e. they should identify that violations can occur): static compliance with such a specification would in fact be evidence in favor of buggy behaviour. On the other hand methods based on runtime verification allow us to monitor for and dealing these violations, without changing the behaviour of the smart contract. This is essential for smart contracts that need to synchronize with other uncontrollable systems. contractLarva is an ideal tool for this, allowing us to detect these violations at runtime.

Not all smart contracts require synchronicity with other systems, but may instead be standalone systems. In this case we may be able to check for compliance with a specification using static code analysis. Runtime verification is also an option, however gas concerns come into play. Consider that given a smart contract that implements a certain specification, and we verify this at runtime by interleaving the same smart contract with the specification, then we are replicating work. This replication may allow us to identify certain bugs in the implementation, which is useful. While replication of the same logic makes the system more robust and trustworthy. However replication increases the amount of gas used by transactions. Using a smart contract instrumented with a specification for **testing** purposes can also be useful and is a situation where gas concerns do not come into play.

Another approach is to use contractLarva as part of the software engineering process, in a **model-driven development** manner. In other words, a developer can use contractLarva to synthesize automatically certain business logic from a specification. This ensures that the specification is being **enforced at runtime**, that is the instrumented smart contract is correct[1].

Another interesting use case involves **proxy smart contracts**. Consider that a deployed smart contract is immutable, however mutability can be simulated by introducing a proxy smart contract that maintains a record of the current implementation address and passes any function call to that address. In this way the behaviour of the proxy smart contract wholly depends on which address its implementation variable points to at runtime, which can be changed. contractLarva can be used to enforce a certain specification on the proxy smart contract, ensuring that although the implementation changes at runtime its behaviour still remains within certain boundaries.

2 Use Cases

2.1 Use Case Repository

The use cases we consider can all be found at `https://github.com/gordonpace/contractLarva/tree/master/use-cases`. They all follow the following file structure:

1. `<name>/<name>.sol`: The smart contract implementing some business logic.
2. `<name>/<name>Spec.dea`: A specification of an aspect of the business logic, possibly including some business logic transforming the behaviour of the smart contract.
3. `<name>/<name>Monitored.sol`: The smart contract after being instrumented/wrapped with the specification.

2.2 Monitoring for Real-World Violations in a Courier Service

`<name>: CourierService`

[1] Modulo the correctness of the implementation of contractLarva and of the used Solidity compiler.

Given the immutable nature of blockchain transactions, a smart contract can be used to allow for dependable record-keeping. This can be to keep track of orders delivered by a courier service. Listing. 1 illustrates such a smart contract, where an order and delivery can be recorded to the blockchain using appropriate functions.

```
contract CourierService{
  bool ordered;
  bool delivered;

  function order(uint _eta, address _buyer, string memory _address) public{
    require(!ordered);
    ordered = true;
  }

  function deliver(address _signer, string memory _address) public{
    require(!delivered);
    delivered = true;
  }
}
```

Listing 1: Courier Service smart contract.

This smart contract only does basic validation, allowing an order to be ordered only once, and a delivery to be delivered only once.

CONTRACTLARVA can be used to specify more sophisticated business logic, e.g. in Listing. 2 we specify that an order cannot be delivered before being ordered (see lines 17 and 23), and that an order should be delivered within the expected estimated time of arrival, and at the appropriate address (see lines 20 and 25).

```
monitor CourierService{

  declarations {
    string orderAddress;
    uint orderETA;

    function stringEquality (string memory a, string memory b) public view
        returns (bool) {
      return (keccak256(abi.encodePacked((a))) == keccak256(abi.encodePacked((b))) );

    }
  }

  DEA NoDeliveryBeforeOrder {
    states {
      Start: initial;
      Ordered;
      Bad: bad;
      Good: accept;
    }

    transitions {
      Start -[after(order(_eta, _buyer, _address))
            ~> orderAddress = _address; orderETA = _eta;]-> Ordered;

      Ordered -[after(deliver(_signer, _address))
            | orderETA <= now && stringEquality(_address, orderAddress)]-> Good;

      Start -[after(deliver(_signer, _address))]-> Bad;

      Ordered -[after(deliver(_signer, _address))
            | orderETA > now || !stringEquality(_address, orderAddress)]-> Bad;
    }
  }
}
```

Listing 2: Monitor that only ordered items are delivered.

In this use case we are allowing for violating behaviour to be written to the blockchain, instead of failing with a `require`, since we want the smart contract to reflect the state of the real world. Instead of reverting a violation a user may query the monitored smart contract to check whether it is in a bad state.

A limitation of this use case is that we are limiting ourselves to only one order, rather than multiple orders. DEAs currently lack the power to specify the behaviour of each order in a multi-order smart contract. We intend to extend DEAs with this notion of *typestate* in the future.

1. *This contract is between ⟨buyer-name⟩, henceforth referred to as 'the buyer' and ⟨seller-name⟩, henceforth referred to as 'the seller'. The contract will hold until either party requests its termination.*

2. *The buyer is obliged to order at least ⟨minimum-items⟩, but no more than ⟨maximum-items⟩ items for a fixed price ⟨price⟩ before the termination of this contract.*

3. *Notwithstanding clause 1, no request for termination will be accepted before ⟨contract-end-date⟩. Furthermore, the seller may not terminate the contract as long as there are pending orders.*

4. *Upon enactment of this contract, the buyer is obliged to place the cost of the minimum number of items to be ordered in escrow.*

5. *Upon accepting this contract, the seller is obliged to place the amount of ⟨performance-guarantee⟩ in escrow, otherwise, if only a partial amount is placed, the seller is obliged to place the rest by a time period at the buyer's discretion.*

6. *While the contract has not been terminated, the buyer has the right to place an order for an amount of items and a specified time-frame as long as (i) the running number of items ordered does not exceed the maximum stipulated in clause 2; and (ii) the time-frame must be of at least 24 hours, but may not extend beyond the contract end date specified in clause 2.*

7. *Upon placing an order, the buyer is obliged to ensure that there is enough money in escrow to cover payment of all pending orders.*

8. *Before termination of the contract, upon delivery the seller must receive payment of the order.*

9. *Upon termination of the contract, if either any orders were undelivered or more than 25% of the orders were delivered late, the buyer has the right to receive the performance guarantee placed in escrow according to clause 5.*

Figure 1: A legal contract regulating a procurement process.

2.3 Enforcing a Real-World Procurement Contract

`<name>: Procurement`

A use case for blockchains and smart contracts is as the theatre wherein parties to a real-world contract interact. This allows us to enforce some aspects of the real-world contract.

In this use-case we consider a procurement contract, i.e. a contract wherein a buyer binds themselves to buy some amount of goods from a seller, at a certain price and by a certain time.

Consider that the seller has presented a legal contract to the buyer, along with a smart contract which the buyer claims only allows the behaviour specified in the real-world contract. Figure. 1 specifies such a legal contract, while Listing. 3 is an extract from such a smart contract.

```
pragma solidity ^0.4.15;


contract Procurement {
  enum ContractStatus { Proposed, Open, Closed }
  enum OrderStatus { Ordered, Delivered }

  struct Order {
    bool exists;
    uint cost;
    OrderStatus status;
    uint deliveryTimeDue;
  }

...

  function acceptContract() public payable bySeller {
    require(msg.value >= performanceGuarantee);
    contractStatus = ContractStatus.Open;
  }

  function createOrder(
    uint8 _orderNumber,
    uint8 _orderSize,
    uint _orderDeliveryTimeLeft
  ) public payable byBuyer
  {
    // Order does not already exist
    require(!orders[_orderNumber].exists);
    // Number of items ordered does not exceed maximum
    require(itemsOrderedCount + _orderSize <= maximumItemsToBeOrdered);
    // Order delivery deadline will not be too late
    require(now + _orderDeliveryTimeLeft <= endOfContractTimestamp);

    // Ensure there is enough money left in the contract to pay for the order
    uint orderCost = _orderSize * costPerUnit;
    moneyLeftInContract += msg.value;
    require(orderCost <= moneyLeftInContract);
    moneyLeftInContract -= orderCost;

    // Update number of items ordered
```

```
42      itemsOrderedCount += _orderSize;
43
44      // Update contract status
45      pendingOrderCount++;
46      pendingOrderCost += orderCost;
47
48      // Record the order
49      orders[_orderNumber] = Order(true, orderCost, OrderStatus.Ordered, now+
            _orderDeliveryTimeLeft);
50    }
51
52  }
```

Listing 3: Excerpt from procument smart contract.

Checking the smart contract is always compliant with the required behaviour is a hard problem. Instead, CONTRACTLARVA can be used to enforce the behaviour on the execution trace at runtime, simply through building a formal representation of the legal contract.

For example, Listing. 4 specifies formally the second clause of Figure. 1. Consider that with Line 20 we are keeping count of the number of orders, while in Line 21 (Line 22) we are ensuring that the minimum (maximum) items to be ordered is constant and is not modified while the contract is running. With Lines 23 and 24 we ensure that just before the contract is terminated enough items have been ordered, otherwise termination is reverted (note Lines 6-8).

```
1  monitor Procurement{
2    declarations {
3      uint orderCount;
4    }
5
6    reparation {
7      revert();
8    }
9
10   //The buyer is obliged to order at least <minimum-items>, but no more than <maximum-items>
         items for a fixed price <price> before the termination of this contract.
11   DEA EnoughItemsOrdered{
12     states{
13       DuringContract: initial;
14       RangesChanged: bad;
15       OutsideOfRange: bad;
16       EnoughItems: accept;
17     }
18
19     transitions{
20       DuringContract -[after(createOrder(_orderNumber, _orderSize, _orderDeliveryTimeLeft))
             | ~> orderCount += _orderNumber;]-> DuringContract;
21       DuringContract -[minimumItemsToBeOrdered@(LARVA_previous_minimumItemsToBeOrdered !=
             minimumItemsToBeOrdered)]-> RangesChanged;
22       DuringContract -[maximumItemsToBeOrdered@(LARVA_previous_minimumItemsToBeOrdered !=
             minimumItemsToBeOrdered)]-> RangesChanged;
23       DuringContract -[after(terminateContract()) | orderCount < minimumItemsToBeOrdered ||
             orderCount > maximumItemsToBeOrdered]-> OutsideOfRange;
24       DuringContract -[after(terminateContract()) | orderCount >= minimumItemsToBeOrdered &&
              orderCount <= maximumItemsToBeOrdered]-> EnoughItems;
25     }
26   }
27 }
```

Listing 4: Monitor specification that checks whether the number of items ordered is within the minimum and maximum required.

The buyer can request the smart contract be instrumented with such a DEA to ensure they can trust the smart contract at runtime.

Moreover, DEAs can be used to deal with possible edge cases. For example Listing. 5 specifies that any ether left in the contract after termination should be transferred to the contract's deployer. Note how Lines 8-10 ensure that when a contract is initialised we keep track of the contract's deployer in the `mediator` variable. Line 25 specifies that if the contract ends with some balance then we can transition to a bad state, which activates the reparation specified by Lines 12-14, i.e. that the mediator is transferred the remaining balance.

```
1    monitor Procurement{
2      declarations {
3        address mediator;
4        uint orderCount;
5        bool noEtherAfterTerminationBadStateReached;
6      }
7
8      initialisation {
9        mediator = address(uint160(msg.sender));
```

```
10      }
11
12      reparation {
13        noEtherAfterTerminationBadStateReached ? mediator.transfer(this.balance) : ();
14      }
15
16      //When a contract terminates there should not be any ether left in its balance.
17      DEA NoEtherAfterTermination{
18        states{
19          DuringContract: initial;
20          EndedWithBalance: bad;
21          EndedWithoutBalance: accept;
22        }
23
24        transitions{
25          DuringContract -[after(terminateContract) | this.balance != 0 ~>
                noEtherAfterTerminationBadStateReached = true;]-> EndedWithBalance;
26          DuringContract -[after(terminateContract) | this.balance == 0]->
                EndedWithoutBalance;
27        }
28      }
```

Listing 5: Monitor that checks that when the procurement contract terminates no ether is left in the smart contract.

2.4 Enforcing an ERC20 Specification

`<name>: FixedSupplyToken`

A common use for blockchains is to manage cryptocurrencies. In fact smart contracts are used to implement wallets (usually following some conventional interface such as ERC20 in Listing. 6), allowing users to own a certain amount of tokens and to use them. The design and implementation of these wallets is critical, in fact in the past bugs in wallet implementations have led to a significant amount of tokens with real-world value being lost (e.g. the Parity bug). CONTRACTLARVA monitors implementing aspects of the expected behaviour of a wallet can serve as a further defense against such bugs and other attacks.

```
1       // ----------------------------------------------------------------------------
2       // ERC Token Standard #20 Interface
3       // https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md
4       // ----------------------------------------------------------------------------
5       interface ERC20TokenImplementation {
6        function totalSupply () external constant returns (uint);
7        function balanceOf (address tokenOwner) external constant returns (uint balance);
8        function allowance (address tokenOwner, address spender) external constant returns
             (uint remaining);
9        function transfer (address caller, address to, uint tokens) external returns (bool
             success);
10       function approve (address caller, address spender, uint tokens) external returns (
             bool success);
11       function transferFrom (address caller, address from, address to, uint tokens)
             external returns (bool success);
12       event Transfer (address indexed from, address indexed to, uint tokens);
13       event Approval (address indexed tokenOwner, address indexed spender, uint tokens);
14
15      }
```

Listing 6: ERC20 interface.

Here we consider a fixed supply token wallet, where the wallet is initialised with a certain amount of tokens that should not change at runtime. Looking at the implementation of the `transfer` function in Listing. 7, we can see that the balance of the function caller is reduced on Line 2, and the balance of the intended recipient is increased on Line 3. Note that here addition and subtraction are encoded in appropriate functions, so that CONTRACTLARVA is able to instrument them.

```
1   function transfer(address to, uint tokens) onlyOwner public returns (bool success) {
2       balances[msg.sender] = sub(balances[msg.sender],tokens);
3       balances[to] = add(balances[to],tokens);
4       emit Transfer(caller, to, tokens);
5       return true;
6   }
```

Listing 7: `transfer` function.

One way to ensure there is a fixed supply of tokens is to simply check that any addition to any user's balance is paired with a subtraction from another user's balance. Listing. 8 encodes this logic.

```
1  monitor FixedSupplyToken{
2
3      declarations{
4          uint currentTokens;
5      }
6
7      reparation{
8          revert();
9      }
10
11     //This property checks that any addition to the balance must be coupled immediately with
           a subtraction,
12     //ensuring tokens are only moved around, while ensuring that any change is immediately (
           modulo one step) reflected in the total sum.
13     DEA AdditionOfBalanceMustBeAccompaniedBySubtraction{
14         states{
15             Before: initial;
16             StartTransfer;
17             SubAfterAdd;
18             AddAfterSub;
19             EndTransfer;
20             UnMatchedModification: bad;
21         }
22
23         transitions{
24             Before -[before(transfer(caller, to, tokens)) | ~> currentTokens = tokens;]->
                   StartTransfer;
25
26             StartTransfer -[after(add(a, tokens)) | currentTokens == tokens]-> SubAfterAdd;
27             StartTransfer -[after(add(a, tokens)) | currentTokens != tokens]->
                   UnMatchedModification;
28             StartTransfer -[after(sub(a, tokens)) | currentTokens == tokens]-> AddAfterSub;
29             StartTransfer -[after(sub(a, tokens)) | currentTokens != tokens]->
                   UnMatchedModification;
30
31             SubAfterAdd -[after(sub(a, tokens)) | currentTokens == tokens]-> EndTransfer;
32             SubAfterAdd -[after(sub(a, tokens)) | currentTokens != tokens]->
                   UnMatchedModification;
33             SubAfterAdd -[after(add(a, tokens))]-> UnMatchedModification;
34
35             SubAfterAdd -[after(add(a, tokens)) | currentTokens == tokens]-> EndTransfer;
36             SubAfterAdd -[after(add(a, tokens)) | currentTokens != tokens]->
                   UnMatchedModification;
37             SubAfterAdd -[after(sub(a, tokens))]-> UnMatchedModification;
38
39             EndTransfer -[after(transfer(caller, to, tokens)) | ~> currentTokens = 0;]->
                   Before;
40             SubAfterAdd -[after(transfer(caller, to, tokens))]-> UnMatchedModification;
41             AddAfterSub -[after(transfer(caller, to, tokens))]-> UnMatchedModification;
42         }
43     }
44 }
```

Listing 8: Monitor that checks that an addition in tokens is accompanied with an equal subtraction.

Consider that the DEA is a functional specification for the `transfer` function. That is, it is activated when a transfer starts (Line 24), and should give a verdict or restart after the transfer ends (Line 39). Consider also that an appropriate implementation may either first reduce the balance of the sender and the increase the balance of the recipient, or vice-versa. Similarly the specification is symmetric (see states on Line 17 and Line 18, and transitions on Lines 26-37). The specification first keeps track of the number of tokens to be transferred (see the variable `currentTokens` declared on Line 4 and the action part of the transition on Line 24). Then, any addition or subtraction must be equivalent to this value (see Line 26), while any divergence leads to a violation (see Line 27). If the transfer ends before a pair of addition and subtraction operations are performed then th monitor ends in a bad state (see Lines 40 and 41).

2.5 Enforcing a Casino Specification

`<name>: Casino`

Smart contracts have been used to allow for people to bet a certain amount of crypto-currency tokens, which allows for users who guessed the correct outcome to be automatically rewarded. This use case considers such a smart contract, which can be found in Listing. 9.

```
1 pragma solidity ^0.5.11;
2
3 contract Casino{
4
5     mapping(uint => mapping (uint => address payable[])) placedBets;
```

```
6     mapping(uint => mapping(address => uint)) potShare;
7
8     uint[] numbersGuessed;
9
10    uint pot;
11
12    uint tableID;
13    uint tableOpenTime;
14
15    address owner;
16
17    constructor() public{
18        owner = msg.sender;
19    }
20
21    function openTable() public{
22        require(msg.sender == owner);
23        require(tableOpenTime == 0);
24
25        tableOpenTime = now;
26        tableID++;
27    }
28
29    function closeTable() public{
30        require(msg.sender == owner);
31        require(pot == 0);
32
33        delete numbersGuessed;
34    }
35
36    function timeoutBet() public{
37        require(msg.sender == owner);
38        require(now - tableOpenTime > 60 minutes);
39        require(pot != 0);
40
41        for (uint i = 0; i < numbersGuessed.length; i++) {
42            uint l = placedBets[tableID][numbersGuessed[i]].length;
43
44            for (uint j = 0; j < l; j++) {
45                address payable better = placedBets[tableID][numbersGuessed[i]][l];
46                better.transfer(potShare[tableID][better]);
47                delete placedBets[tableID][numbersGuessed[i]];
48            }
49        }
50
51        closeTable();
52    }
53
54    function placeBet(uint guessNo) payable public{
55        require(msg.value > 1 ether);
56
57        potShare[tableID][msg.sender] += msg.value;
58        placedBets[tableID][guessNo].push(msg.sender);
59        numbersGuessed.push(guessNo);
60        pot += msg.value;
61    }
62
63    //we assume owner is trusted
64    function resolveBet(uint _secretNumber) public{
65        require(msg.sender == owner);
66
67        uint l = placedBets[tableID][_secretNumber].length;
68        if(l != 0){
69            for (uint i = 0; i < l; i++) {
70                placedBets[tableID][_secretNumber][i].transfer(pot/l);
71            }
72        }
73
74        pot = 0;
75
76        closeTable();
77    }
78 }
```

Listing 9: Casino smart contract.

A user may not trust such a smart contract enough to handle their money. The owner may increase such confidence by instrumenting the smart contract with appropriate specifications, while the user may also instrument the smart contract and test the resulting specification on a testnet.

In Listing. 10 we consider two properties the casino should have: (i) when there is an ongoing bet (i.e. the table is open) the running pot should not be reduced but only increase; and (ii) the table cannot be closed during a bet.

```
1 monitor Casino{
```

```
2
3     declarations{
4         uint total;
5     }
6
7     DEA NoReduction {
8         states {
9             TableOpen: initial;
10            TableClosed: accept;
11            BetPlaced;
12            PotReduced: bad;
13        }
14        transitions {
15            TableOpen -[after(closeTable) | pot == 0 ]-> TableClosed;
16            TableOpen -[after(placeBet(_value)) | _value <= pot ~> total += _value;]->
                  BetPlaced;
17            BetPlaced -[after(timeoutBet)]-> TableOpen;
18            BetPlaced -[after(resolveBet)]-> TableOpen;
19            BetPlaced -[pot@(LARVA_previous_pot > pot)]-> PotReduced;
20        }
21    }
22
23    DEA OpenUntilResolution {
24        states {
25            TableClosed: initial;
26            TableOpen;
27            BetPlaced;
28            TableCloseDuringBet: bad;
29        }
30        transitions {
31            TableClosed -[after(openTable)]-> TableOpen;
32            TableOpen -[after(closeTable)]-> TableClosed;
33            TableOpen -[after(placeBet)]-> BetPlaced;
34            BetPlaced -[after(resolveBet)]-> TableOpen;
35            BetPlaced -[after(timeoutBet)]-> TableOpen;
36            BetPlaced -[after(closeTable)]-> TableCloseDuringBet;
37        }
38    }
39 }
```

Listing 10: Monitors that check: (i) that a game's `pot` is increasing; and (ii) a game is open until it is resolved.

2.6 Safe Mutability of an ERC20 Wallet

`<name>: ERC20Interface`

Smart contracts are immutable, however a proxy design pattern can be used to simulate mutability, by having a proxy smart contract act as the main entry-point, which passes on function calls to different versions of the smart contract implementing the main business logic. This can be unsafe, since the business logic may mutate without any notice. A use-case for ContractLarva is to limit the mutability of the business logic by enforcing a certain specification on the proxy smart contract.

Consider an ERC20 wallet, which has the interface specified in Listing. 6. A proxy smart contract would be similar to the smart contract extract in Listing. 11.

```
1   contract ERC20Interface{
2
3     ERC20TokenImplementation impl;
4     address owner;
5
6     constructor(ERC20TokenImplementation _impl, address _owner) public{
7       impl = _impl;
8       owner = _owner;
9     }
10
11    function updateImplementation(address newImpl) public{
12      require(msg.sender == owner);
13      impl = ERC20TokenImplementation(newImpl);
14    }
15
16    function transfer(address to, uint tokens) public returns (bool success){
17      return impl.transfer(msg.sender, to, tokens);
18    }
19    ...
20  }
```

Listing 11: Extract from ERC20 proxy interface.

The DEA in Listing. 12, when weaved into Listing. 11 enforces the well-behaviour of the `transfer` function. In effect this sets pre- and post-conditions for calls of the function, and disallow re-entrancy

into the proxy smart contract from the concrete implementation. We can create similar specifications for the other non-pure functions.

```
monitor ERC20Interface{
  declarations {
    uint transferPreFrom;
    uint transferPreTo;

    uint transferFromPreFrom;
    uint transferFromPreTo;
    uint preAllowance;

  }

  reparation {
    revert();
  }

  DEA TransferWellBehaviour {
    states {
      Before: initial;
      After;
      Bad: bad;
    }

    transitions {

      Before -[before(transfer(to, tokens)) | ~> {transferPreFrom = balanceOf(msg.sender);
           transferPreTo = balanceOf(to);}]-> After;

      After -[before(transfer(to, tokens))]-> Bad;

      After -[after(transfer(to, tokens)) | transferPreFrom < tokens && (balanceOf(msg.
          sender) != transferPreFrom || balanceOf(to) != transferPreTo)]-> Bad;

      After -[after(transfer(to, tokens)) | transferPreFrom < tokens && (balanceOf(msg.
          sender) == transferPreFrom && balanceOf(to) == transferPreTo)]-> Before;

      After -[after(transfer(to, tokens)) | transferPreFrom >= tokens && (balanceOf(msg.
          sender) != (transferPreFrom - tokens) || balanceOf(to) != (transferPreTo -
          tokens))]-> Bad;

      After -[after(transfer(to, tokens)) | transferPreFrom >= tokens && (balanceOf(msg.
          sender) == (transferPreFrom - tokens) && balanceOf(to) == (transferPreTo -
          tokens))]-> Before;
    }
  }
}
```

Listing 12: Monitor that ensures well-behaviour of `transfer` function with appropriate pre- and post-conditions.

2.7 Adding Insurance Logic to a Courier Service

`<name>: InsuredCourierService`

We have used CONTRACTLARVA to enforce specifications by reverting bad behaviour, however we can go a step further by using it to add more sophisticated logic.

A particular use-case is the addition of insurance logic. Consider the courier service contract in Listing. 13, extended with a function `complain`, wherein the buyer can complain if the order has not yet been delivered.

```
contract CourierService{
  bool ordered;
  bool delivered;
  uint value = 1 ether;
  address buyer;

  function order(uint _eta, address _buyer, string memory _address) public{
    require(!ordered && msg.value == value);
    ordered = true;
    buyer = _buyer;
  }

  function deliver(address _signer, string memory _address) public{
    require(!delivered);
    delivered = true;
  }

  function complain() public{
    require(msg.sender == buyer && !delivered);
```

```
20    }
21  }
```

Listing 13: Courier service smart contract extended with a `complain` function.

In this form, the smart contract does not give any assurances about when the order will be delivered. contractLarva can be used to encode such assurances.

The DEA in Listing. 14 inserts some logic before the smart contract is initialised, where it requires the `payStake` function to be called before the smart contract is enabled. Consider how if the smart contract is not delivered on time this logic pays the customer an amount of ether (specified on Line 8) to make up for the inconvenience.

```
1  monitor CourierService {
2
3    declarations {
4      uint orderedTime;
5      uint minimumInsuredDeliveryTime = 24*30 hours;
6
7      address payable private insurer_address;
8      function getStake() private returns(uint) { return value; }
9      function getInsurer() private returns(address payable) { return insurer_address; }
10     function getInsured() private returns(address payable) { return customer; }
11
12     enum STAKE_STATUS { UNPAID, PAID }
13     STAKE_STATUS private stake_status = STAKE_STATUS.UNPAID;
14
15     function payStake() payable public{
16       require (stake_status == STAKE_STATUS.UNPAID);
17       require (msg.value == getStake());
18       require (msg.sender == getInsurer());
19       stake_status = STAKE_STATUS.PAID;
20       LARVA_EnableContract();
21     }
22   }
23
24   initialisation {
25     insurer_address = msg.sender;
26   }
27
28   reparation {
29     getInsured().transfer(getStake());
30     LARVA_DisableContract();
31   }
32
33   satisfaction {
34     getInsurer().transfer(getStake());
35   }
36
37   DEA UnDelivered{
38     states{
39       Start: initial;
40       Ordered;
41       Delivered: accept;
42       Undelivered: bad;
43     }
44
45     transitions{
46       Start -[after(order) | ~> orderedTime = now;]-> Ordered;
47       Ordered -[after(deliver) | now - orderedTime <= minimumInsuredDeliveryTime]->
             Delivered;
48       Ordered -[after(deliver) | now - orderedTime >= minimumInsuredDeliveryTime]->
             Undelivered;
49     }
50   }
51 }
```

Listing 14: Monitor that implements state-based insurance logic.

2.8 Adding Fail-safe logic to a Multi-Owner Wallet

`<name>: MultiOwnersWallet`

A wallet may be owned by multiple entities rather than by a single owner, requiring sophisticated logic to orchestrate the voting process between the multiple owners.

Figure.15 is an extract of a multi-owner wallet, including a modifier that ensures all owners have signed off on a certain action (Line 6), a function to propose a transaction (Line 18), a transfer function that performs a transaction if all owners sign off on it (Line 24), and a function which owners can use to vote to remove owners (Line 28).

```
pragma solidity ^0.5.11;

contract MultiOwners {
...

    modifier allOwners (uint _id){
        require(owners[msg.sender]);
        actionSignOffs[_id][msg.sender] = true;

        for(uint i = 0 ; i < ownerList.length; i++){
            if(ownerList[i] != address(0) && !actionSignOffs[_id][ownerList[i]]){
                return;
            }
        }
        _;
    }

    function proposeTransaction(address payable _to, uint _val) public anyOwner{
        idTo[id] = _to;
        idVal[id] = _val;
        id++;
    }

    function transfer(uint _id) allOwners(_id) public{
        idTo[_id].transfer(idVal[_id]);
    }

    function removeOwner(address _address) anyOwner public{
        votesToRemove[_address][msg.sender] = true;
        votesToRemoveKeys[_address].push(msg.sender);

        uint countInFavour = 0;
        uint totalCount = ownerCount;

        for(uint i = 0; i < totalCount; i++){
            if(votesToRemove[_address][votesToRemoveKeys[_address][i]]){
                countInFavour++;
            }
        }

        uint limit = 2*totalCount/3;

        if(countInFavour >= limit){
            owners[_address] = false;
            ownerCount--;
            for(uint i = 0; i < ownerList.length; i++){
                if(ownerList[i] != address(0) && ownerList[i] == _address){
                    ownerList[i] = address(0);
                }
            }
        }
    }
}
```

Listing 15: A wallet smart contract allowing for multiple owners.

This smart contract represents the base functionality of the wallet, with little validation. Instead the well-behaviour of the smart contract can be specified separately using DEAs.

In Listing. 16 we specify this well-behaviour, in terms of two DEAs. First we ensure that transactions started by ex-owners are not carried out, and secondly we ensure that an owner can only vote once in removing an owner, since the current implementation allows that.

```
monitor MultiOwners {

    declarations{
        mapping(uint => address) idRequestedBy;
        mapping(address => mapping(address => bool)) votes;
    }

    reparation {
        revert();
    }

    DEA IgnoreTransactionsStartedByExOwners{
        states{
            BeforeTransfer: initial;
            BadTransfer: bad;
        }

        transitions{
            BeforeTransfer -[after(proposeTransaction(_to,_val)) | ~> idRequestedBy[--id]
                = msg.sender;]-> BeforeTransfer;
            BeforeTransfer -[before(transfer(_id)) | !owners[idRequestedBy[id]]]->
                BadTransfer;
        }
```

```
22      }
23
24      DEA NeutraliseDoubleVote{
25          states{
26              BeforeVote: initial;
27              DoubleVote: bad;
28          }
29
30          transitions{
31              BeforeVote -[after(removeOwner(_address)) | ~> votes[_address][msg.sender] =
                    true;]-> BeforeVote;
32              BeforeVote -[before(removeOwner(_address)) | votes[_address][msg.sender]]->
                    DoubleVote;
33          }
34      }
35  }
```

Listing 16: A monitor that checks that transactions started by ex-owners are not fulfilled, and that owners can only vote once.

2.9 Adding Logic to an Auction House

`<name>: SmartAuctionHouse`

Smart contracts are used to automate some part of a real-world process, ensuring that part of the process is carried according to a strict set of rules. An auction is an example of such a process that can be carried out using a smart contract and that we consider in this use case.

Listing. 17 is an implementation of an auction house, allowing auctions to be started (Line 33), allowing people to make offers (Line 48), the auctioneer to start calling the auction (Line 40), and to declare a winning offer (Line 56). A winner can then fulfill their offer (Line 63).

```
1 pragma solidity ^0.4.24;
2
3 contract SmartAuctionHouse{
4
5     uint currentItem;
6     uint startingOffer;
7
8     uint currentOffer;
9     address currentWinner;
10
11    uint ticks;
12
13    mapping(uint => address) winners;
14    mapping(uint => uint) winningOffer;
15    mapping(uint => bool) fulfilled;
16
17    address owner;
18
19    function SmartAuctionHouse(){
20        owner = msg.sender;
21    }
22
23    modifier onlyOwner(){
24        require(msg.sender == owner);
25        _;
26    }
27
28    modifier onlyOwnerOrInternal(){
29        require(msg.sender == owner || msg.sender == address(this));
30        _;
31    }
32
33    function auctionOffItem(uint _offerID, uint _startingOffer) public {
34        require(!ongoingAuction());
35
36        currentItem = _offerID;
37        startingOffer = _startingOffer;
38    }
39
40    function tick() public onlyOwner{
41        require(ongoingAuction());
42
43        ticks++;
44
45        if(ticks > 2) declareWinningOffer();
46    }
47
48    function makeOffer(uint _offer) public {
49        require(_offer > currentOffer);
```

```
50
51          currentOffer = _offer;
52          currentWinner = msg.sender;
53      }
54
55      function declareWinningOffer() public onlyOwnerOrInternal{
56          require(ticks > 2);
57
58          winners[currentItem] = currentWinner;
59          winningOffer[currentItem] = currentOffer;
60          reset();
61      }
62
63      function fulfillOffer(uint _id) payable public {
64          require(winners[_id] == msg.sender && winningOffer[_id] == msg.value && !fulfilled[
                _id]);
65          fulfilled[_id] = true;
66      }
67
68      function ongoingAuction() internal returns(bool){
69          return startingOffer == 0 && currentOffer == 0;
70      }
71
72      function reset() internal {
73          currentItem = 0;
74          startingOffer = 0;
75          currentOffer = 0;
76          currentWinner = address(0);
77          ticks = 0;
78      }
79
80      function getItemWinningOffer(uint _id) public returns(address,uint){
81          return (winners[_id], winningOffer[_id]);
82      }
83
84      function getItemWinningBidder(uint _id) public returns(address){
85          (address bidder, ) = getItemWinningOffer(_id);
86          return bidder;
87      }
88
89      function getItemWinningOffer(uint _id) public returns(uint){
90          (, uint winningOffer) = getItemWinningOffer(_id);
91          return winningOffer;
92      }
93  }
```

Listing 17: A auction house smart contract.

In Listing. 18 we define monitor specifications that check: (i) that only one auction is held at a time, while any auction is automatically ended if fifteen minutes have passed since the last offer; and (ii) any winning bidder is obliged to fulfill their offer by canceling their offer if they attempt to bid more than three times.

For the first property (Lines 42-56), we transition to the `AuctionStart` start when an auction starts (Line 50), while anytime an offer is made the last offer time variable (declared on Line 9) is updated (Line 51). When a winner is declared the monitor transitions back to the initial state (Line 52). Any attempt to start an auction while another is running and the time since last offer is less than fifteen minutes causes a transition to a bad state (Line 53), which means the call fails (Lines 36-38). If however the time since the last offer is more or equal to fifteen minutes the contract is forced to declare a winner (see Lines 54 and Lines 11-15).

For the second property (Lines 61-75), we only have an initial and bad state. For this property we require a mapping to keep track of unfulfilled offers (defined on Line 4). When a winning offer is declared the offer is marked as unfulfilled (Line 68), while upon fulfillment it is marked as fulfilled (Line 69). When a bidder makes an offer and has no unfulfilled bids (checked with the function defined on Lines 25-30) then the offer is allowed treated normally (Line 70), while if there are unfulfilled offers and the bid attempt counter for the bidder is less than three, the bid attempt counter for the bidder is increased (Line 71). If it is equal or more than three then any unfulfilled bids are canceled and the bid canceled (see Line 72, using the function defined on Lines 17-22). If the user attempts to fulfill a bid that has been canceled the property marks a violation (see Line 73) and cancels the transaction (Lines 36-38).

```
1 monitor SmartAuctionHouse{
2
3     declarations{
4         mapping(address => uint) attemptsBeforeFullfillment;
5         mapping(uint => bool) cancelledItems;
6         mapping(uint => bool) unfulfilled;
7         mapping(address => uint[]) wonBids;
```

```
8
9           uint timeSinceLastOffer;
10
11          function forceDeclareWinner() private{
12              ticks = 3;
13              declareWinningOffer();
14              timeSinceLastOffer = 0;
15          }
16
17          function cancelAnyUnfulfilledBids(address _bidder) private{
18              for(uint i = wonBids[_bidder].length - 1; i >= 0; i--){
19                  if(unfulfilled[wonBids[_bidder][i]]){
20                      cancelledItems[wonBids[_bidder][i]] = false;
21                  }
22              }
23          }
24
25          function areAnyUnfullfilled(address _bidder) private returns(bool){
26              for(uint i = wonBids[_bidder].length - 1; i >= 0; i--){
27                  if(unfulfilled[wonBids[_bidder][i]]){
28                      return true;
29                  }
30              }
31
32              return false;
33          }
34      }
35
36      reparation{
37          revert();
38      }
39
40      //auctionOffItem cannot occur subsequently without declareWinningOffer in between
41      //if 15 minutes have passed since the last offer then automatically declare the winner
42      DEA OneAuctionAtATime{
43          states{
44              NoOngoingAuction: initial;
45              AuctionStart;
46              AuctionAttempted: bad;
47          }
48
49          transitions{
50              NoOngoingAuction -[after(auctionOffItem(_offerID, _startingOffer))]->
                    AuctionStart;
51              AuctionStart -[after(makeOffer(_offer)) | ~> timeSinceLastOffer = now;]->
                    AuctionStart;
52              AuctionStart -[after(declareWinningOffer())]-> NoOngoingAuction;
53              AuctionStart -[before(auctionOffItem(_offerID, _startingOffer)) | now -
                    timeSinceLastOffer < 15 minutes]-> AuctionAttempted;
54              AuctionStart -[before(auctionOffItem(_offerID, _startingOffer)) | now -
                    timeSinceLastOffer >= 15 minutes ~> forceDeclareWinner();]-> AuctionStart;
55          }
56      }
57
58      //if winning bid is not fulfilled within a day's time
59      // then the bidder is not allowed to bid on other items
60      //      and any attempt to bid more than 3 times before paying then the winning bid is
            cancelled
61      DEA WinningBidsMustBeFulfilledOrCancelled{
62          states{
63              Initial: initial;
64              UnfulfilledBids: bad;
65          }
66
67          transitions{
68              Initial -[before(declareWinningOffer()) | ~> unfulfilled[currentItem] = true;]->
                     Initial;
69              Initial -[after(fulfillOffer(_id)) | ~> unfulfilled[_id] = false;]-> Initial;
70              Initial -[before(makeOffer(_offer)) | !areAnyUnfullfilled(msg.sender) ~>
                    attemptsBeforeFullfillment[msg.sender] = 0;]-> Initial;
71              Initial -[before(makeOffer(_offer)) | areAnyUnfullfilled(msg.sender) &&
                    attemptsBeforeFullfillment[msg.sender] < 3 ~> attemptsBeforeFullfillment[msg
                    .sender]++;]-> Initial;
72              Initial -[before(makeOffer(_offer)) | areAnyUnfullfilled(msg.sender) &&
                    attemptsBeforeFullfillment[msg.sender] >= 3 ~> cancelAnyUnfulfilledBids(msg.
                    sender); return;]-> UnfulfilledBids;
73              Initial -[before(fulfillOffer(_id)) | cancelledItems[_id]]-> UnfulfilledBids;
74          }
75      }
76  }
```

Listing 18: A monitor that checks that only one auction is ongoing at any point in time and setting time limits on when a winning bid needs to be fulfilled.

2.10 Other

2.10.1 Ensuring Wallet maintains deposit

`<name>: WalletWithDeposit`

We consider a wallet that must be initialised with some ether. Upon the smart contract being self destructed the deposit must be given to a set address, and the owner receives any remaining ether. We enforce with a specification that any transfer from the wallet (except during self-destruct) never results in the deposit leaving the wallet, and that the deposit owner's address is never changed to be the same as that of the owner. Moreover we check that the history variable, of mapping type, always maintains correct information.