Runtime Monitoring of Smart Contracts

On the Ethereum network

Lars Stegeman [s1346466]
l.stegeman@student.utwente.nl

April 12, 2018

Contents

1 Introduction

2 Background

The Ethereum platform is built upon a distributed public ledger . On this ledger the cryptocurrency ether is stored. It is opposed to Bitcoin based on an account based system and not unspend transaction output. There are two types of accounts, one is a default account in which a user controls the spending of funds through its private keys. These accounts are called "Externally owned Accounts". The other option is a "Contract Account', which means that it is managed by code only. The code is set when the contract is constructed and initialised on the blockchain. Contract accounts only execute code when they are called from other contracts. Each contract has a persistent storage which is also maintained on the blockchain. This means that the Ethereum blockchain consists of two parts. The first part is the transaction history and the other part is the storage of all the deployed smart contracts combined. Transactions are the only entity that make changes to the storage. At an higher level overview we could see the Ethereum network as a large state machine in which changes to the state are controlled by transactions. Transactions are grouped in blocks and these blocks are distributed over the network and validated by each node.

2.1 Smart Contracts

Smart Contracts on the Ethereum network consist of two parts. Each contract has a set of functions and a storage. The contract set of functions is defined by the contract code that is deployed with the contract. This contract code is EVM bytecode and is usually compiled from a higher level programming language. When the contract is created the storage is initially empty. Only the contract code can make changes and add data to the presistent storage, within this storage the state of the contract is maintained. Each new function call has an empty memory, this can also be used to store data. But this data is not presistent through transactions, it is only persistent within the transaction. There are also so called "logs", this storage can only be used to store data and not retrieve. This storage is usually used to provide data for the external world because it can be searched efficiently.
Functions are only executed when they are called by external contracts. For example if a fund is to be released after a certain amount of time (block number higher then a certain amount). These funds will not be automatically transferred once the time treshold is reached, they will only be released when the function is called again.

2.2 EVM

2.3 Solidity/Bamboo/Vyper

Smart contracts are usually written in a language that compiles to EVM (Ethereum Virtual Machine) bytecode. Currently the best known and most used language is Solidity. But there are other options available that compile to the same EVM. They differ in their syntax and influences by other languages.

- Solidity `http://solidity.readthedocs.io/en/latest/` is a contract oriented, high-level language for implementing smart contracts. Solidity is statically typed and supports inheritance. Its syntax is influenced by Javascript.
- Bamboo `https://github.com/pirapira/bamboo` is a programming language which makes state transitions explicit. This way it avoids reentrancy by default. Instead of having a global state of the contract, contracts morph into new contract by calling functions. This way there should be less suprises in the exeuction of smart contracts.
- Vyper `https://github.com/ethereum/vyper` is still an experimental programming language. The idea is to limit certain functions and aspects that are possible in Solidity to make writing smart contracts more secure. It also tries to make smart contracts more human readable to make it simpler to see what will happen when a function is called.

3 Example

In this section a small example will be introduced. This example helps understand the problem this research tries to solve. The tool that will parse specification is not implemented yet, that is why in this example the specification is translated by hand. First we will introduce the contract and what is it supposed to do. Next we will add the specifications to the contract that we wish to check for during runtime. After that we will look at the code of the runtime monitored contract, which is made by translating the specified properties to Solidity code and adding it to the existing contract. Lastly we will discuss what this means for the contract in terms of gas execution and additional changes that had to be made to the contract.

3.1 Simple Token

In this example we will use the contract SimpleToken. The contract code can be found on the Ethereum Foundation website [1]. It models a minimum viable token, to keep the state of the contract a `mapping` is used that maps `address` to `uint`. This mapping is kept in the contracts internal storage and is stored on the blockchain. It indicates which addresses holds how much of this

[1] `https://www.ethereum.org/token`

contracts token and changes each time the `transfer` function is called. The `transfer` function requires two parameters an address (`_to`) and an uint(`_value`) which specifies the amount to be sent. The from address is determined from the global variable `msg` which is present in each transaction. When the contract is created the constructor will be called. In this constructor the initialSupply is given as a parameter. All the initial supply is given to the contract creator (`msg.sender`). The totalSupply value is assigned and cannot be changed after initialisation.
Note that this contract is not ERC20 compliant. ERC20 is the interface that most tokens use to implement the desired functionality. This interface is defined in order for all wallets and exchanges to be able to handle different tokens [2]. The main difference is that this contract does not have an `approve` mapping which lets users approve a certain transfer of tokens. Also this SimpleToken does not allow `minting` or `burning` of tokens, in other words the total supply is fixed. Below we can see the Solidity source code of the contract SimpleToken.

```
pragma solidity ^0.4.20;

contract SimpleToken {
  /* This creates an array with all balances */
  mapping (address => uint256) public balanceOf;
  /* TotalSupply is fixed for this token, and does not change. */
  /* It is assigned in the constructor */
  uint256 totalSupply;

  /* Initializes contract with initial supply tokens to the creator of the contract
       */
  function SimpleToken(uint256 initialSupply) public {
    // Give the creator all initial tokens
    balanceOf[msg.sender] = initialSupply;
    totalSupply = initialSupply;
  }

  /* Send coins */
  function transfer(address _to, uint256 _value) public {
    // Check if the sender has enough
    require(balanceOf[msg.sender] >= _value);
    // Check for overflows
    require(balanceOf[_to] + _value >= balanceOf[_to]);
    // Subtract from the sender
    balanceOf[msg.sender] -= _value;
    // Add the same to the recipient
    balanceOf[_to] += _value;
  }
}
```

3.2 Specification

The above section describes the implementation of the token contract. However there is also a specification given in words as to what the contract should do. A few properties of this specification can be declared explicitly using pre and postconditions or invariants. These properties are important to the functionality of the contract. Properties could be translated to corresponding Solidity code and added to the contract. This code is executed within each transaction and thus the properties are checked and validated at runtime.
The first property is that the value of `totalSupply` is always equal to the sum of all the individual balances in the mapping `balanceOf`. Another property is that when a transfer function is executed the balance of the `_to` address is incremented with the `_value`. And the balance of the sender is decreased with the same value. The first property can be seen as an invariant of the contract and should hold before and after execution of a function. The second property should be checked after the execution of the transfer function and can be represented as a postcondition on that function. The exact syntax on how to declare these properties is not defined yet. This example will use syntax that is close to the syntax that is used by JML.

```
pragma solidity ^0.4.20;

contract SimpleToken {
  /* This creates an array with all balances */
  mapping (address => uint256) public balanceOf;
```

[2] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

```
  /* TotalSupply is fixed for this token, and does not change. */
  /* It is assigned in the constructor */
  uint256 totalSupply;


  /* @invariant totalSupply ==  sum = (forall x in balanceOf [ sum +=balanceOf[x]])

  /* Send coins */
  /* @ensures
      balanceOf[_to] == \old(balanceOf[_to]) + _value &&
      balanceOf[msg.sender] == \old(balanceOf[msg.sender]) - _value &&
      forall x : x != _to || x != msg.sender : balanceOf[x] == \old(balanceOf[x])
  */
  function transfer(address _to, uint256 _value) public {
  }
}
```

3.3 Implementation of RuntimeMonitoredSimpleToken

The specification together with the solidity code should be compiled to a new smart contract. The compiled smart contract will have the same behaviour but with extra assertions added to the code. The tool should be able to parse specifications and add the correct code to the corresponding functions. For this example the code will be made by hand because the tool still needs to be developed. The contract solidity code for the Runtime-Monitored SimpleToken can be seen in the snippet below.
For this example if an assertion in the specification is false, the call will return an error. Later more complex behaviour can be added when this situation occurs. The way the contract works from the outside should not change, because several front-end implementations could depend on it. This means that the added behaviour should be inside the functions that are in the original contract. To accomplish this the transfer function body is moved to a seperate private function `transfer_body()`. The original transfer function will do several things. First it stores the current state in the `memory` storage of the contract. This is needed because specifications can use the `\old` keyword to reference to variables before the function execution. Next it checks the invariant and possible preconditions. After that the function body is executed. Lastly the postcondition and invariant are checked, the state before execution is passed as a parameter to the functions.

```
pragma solidity ^0.4.20;

contract RuntimeMonitoredSimpleToken {
  /* This creates an array with all balances */
  mapping (address => uint256) public balanceOf;
  /* TotalSupply is fixed for this token. */
  /* It is assigned in the constructor */
  uint256 totalSupply;


  /* ------------------------------------------------------------ */
  /* Array to keep a list of in use addresses */
  address[] public addressesInUse;
  /* Struct to save the state of the balanceOf mapping */
  struct BalanceOfStruct{
    address _address;
    uint256 balance;
  }
  /* ------------------------------------------------------------ */


  /* Initializes contract with initial supply tokens to the creator of the contract
     */
  function RuntimeMonitoredSimpleToken(uint256 initialSupply) public {
    // Give the creator all initial tokens
    balanceOf[msg.sender] = initialSupply;
    /* ------------------------------------------------------------ */
    addressesInUse.push(msg.sender);
    /* ------------------------------------------------------------ */
    totalSupply = initialSupply;
  }

  /* Send coins */
```

```
function transfer(address _to, uint256 _value) public {
  //Save state of current variables needed for validation of properties
  BalanceOfStruct[] memory  old_balanceOf = new BalanceOfStruct[](addressesInUse.
      length);
  address[] memory old_addressesInUse = new address[](addressesInUse.length);
  for(uint x; x < addressesInUse.length; x++){
    old_addressesInUse[x] = addressesInUse[x];
    old_balanceOf[x] = BalanceOfStruct(addressesInUse[x], balanceOf[
        addressesInUse[x]]);
  }

  // check invariant
  invariant();
  // check requires

  // execute body
  transfer_body(_to,_value);
  // check ensures
  transfer_ensures(_to, _value, old_addressesInUse, old_balanceOf);
  // check invariant
  invariant();
}

function transfer_ensures(address _to, uint256 _value, address[]
    _old_addressesInUse, BalanceOfStruct[] _old_balanceOf ) private{
  uint index_to;
  uint index_from;
  for(uint x; x< addressesInUse.length; x++){
    if(addressesInUse[x] == _to){
      index_to=x;
    }
    if(addressesInUse[x] == msg.sender){
      index_from = x;
    }
  }
  uint old_balance_from = _old_balanceOf[index_from].balance;
  uint old_balance_to ;
  if(index_to >= _old_addressesInUse.length){
      old_balance_to = 0;
  }else{
      old_balance_to = _old_balanceOf[index_to].balance;
  }

  bool exp1 = (balanceOf[_to] == (old_balance_to + _value));
  bool exp2 = (balanceOf[msg.sender] == (old_balance_from - _value));
  bool exp3 = true;
  for(x=0; x < _old_addressesInUse.length && exp3; x++){
    if(x != index_to || x!= index_from){
      exp3 == (balanceOf[addressesInUse[x]] == _old_balanceOf[x].balance);
    }
  }
  assert(exp1 && exp2 && exp3);
}

function invariant() private{
  uint256 sum;
  for(uint x; x < addressesInUse.length; x++){
    sum += balanceOf[addressesInUse[x]];
  }
  assert(totalSupply == sum);
}

/* transfer function original body */
function transfer_body(address _to, uint256 _value) private {
  // Check if the sender has enough
  require(balanceOf[msg.sender] >= _value);
  // Check for overflows
  require(balanceOf[_to] + _value >= balanceOf[_to]);
  // Subtract from the sender
  balanceOf[msg.sender] -= _value;
  // Add the same to the recipient
  balanceOf[_to] += _value;
  /* ------------------------------------------------------------ */
```

```
    bool inUse = false;
    for(uint x; x < addressesInUse.length && !inUse; x++){
      if(addressesInUse[x] == _to){
          inUse = true;
      }
    }
    if(!inUse){
        addressesInUse.push(_to);
    }
    /* ------------------------------------------------------------ */
  }
}
```

3.4 Discussion

There are a few choices that had be made during the implementation of the runtime contract. The first observation is that `mappings` are not iterable. This means that properties that use quantifiers and reference mappings cannot be tested at runtime. Mappings in Solidity do not store a keyset and just store the information at the corresponding hash of the key. This limitation can be avoided if we store the used keyset in an additional array. This array must be kept in the storage since it must be peristent between transactions. The SimpleToken contract has only one mapping and the storage array `addressesInUse` keeps track of the keys. A more general approach would be to define a new storage construction called IterableMapping [3].
Another limitation is that mappings cannot be declared in `memory`. Thus to store the variable `\old(someMapping{ key => value}` there is need for either a additional mapping in `storage` or the mapping can be represented by a struct array (which can be stored in `memory`). The last approach is used in the example, but additional analysis is needed to determine which method is the most gas efficient. Gas cost is significantly higher for `storage` in comparison to `memory`, but additional actions are required to have the information converted to a struct variable.
Lastly because of additional checks the gas cost for each function is not constant anymore. This effect is worse compared to the increase of gas cost in normal contracts. For example if the SimpleToken contract had 1000 addresses in use the cost of each function call would not increase in the original contract. But in the runtime-monitored contract the gas costs would increase since the array `addressesInUse` has length 1000 which causes extra iterations for validation.

4 Runtime monitoring

5 Property specification

6 Related Work

During initial research two runtime verification frameworks were found on Github. Both of them are described in short below. But very little documentation is available for both of them.

6.1 LARVA

LARVA can be found on github at `https://github.com/gordonpace/contractLarva`. From the instructions on the README you can write a specification and a contract in Solidity. The compiler will combine these two and output a new Solidity contract with the runtime verification checks in place.

6.2 Ethereuem-runtime-verification

This project is located at `https://github.com/shaunazzopardi/ethereuem-runtime-verification`. No documentation is available for this project. It mentions the LARVA project in the description in that it differs from LARVA because this runtime-verification tool can dynamically add properties to an already deployed smart contract.

[3] `https://github.com/ethereum/dapp-bin/blob/master/library/iterable_mapping.sol`

7 Planning