Runtime Monitoring of Smart Contracts
On the Ethereum network

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

April 4, 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

To give an exampe we will use the contract SimpleToken. This contract is not ERC20 compliant, but only allows you to transfer coins. It does not have the approve functionality that ERC20 has. Below we will see the solidity code for 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 c
        uint256 totalSupply;

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

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

This token contract has the minimal functionality that a token contract needs. This however is the implementation of the contract. What is missing is the specification of what should happen during exeuction of the contract. There are a few properties that are important to the functionality of the token contract. One 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` addres is incremented with the `_value`. And the same balance of the sender is decreased with the same value. Or when one of the require clauses fails the state is not changed and the complete balance of mapping remains unchanged. The first property can be seen as an invariant and written more formally:
@invariant totalSupply == sum = (forall x in balanceOf [sum +=balanceOf[x]])
The other property should be checked after execution of the transfer function:
@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])
||
forall x: balanceOf[x] == \old(balanceOf[x])

This together with the solidity code could be compiled to a new smart contract. The compiled smart contract will have the same behaviour but with extra assertions added to the code. A tool would be able to parse specifications and add the correct code to the corresponding functions. For this example we will make the code by hand because the tool still needs to be developed.

```
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 c
        uint256 totalSupply;

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

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


}
```

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 avaiable 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.

7 Planning