

# MODEL-BASED TESTING FOR ETHEREUM SMART CONTRACTS



Noureen Jan

Enrollment No: 01-241172-020

A thesis submitted to the Department of Software Engineering, Faculty of Engineering Sciences, Bahria University, Islamabad in the partial fulfillment for the requirements of a Master degree in Software Engineering

June 2020

# Approval Sheet

## Thesis Completion Certificate

Scholar's Name: **Noureen Jan** Registration No: **01-241172-020**  
Program of Study: **MS Software Engineering**  
Thesis Title: **Model Based Testing for Ethereum Smart Contracts**

It is to certify that the above student's thesis has been completed to my satisfaction and, to my belief, its standard is appropriate for submission for Evaluation. I have also conducted plagiarism test of this thesis using HEC prescribed software and found similarity index at \_\_\_\_\_ that is within the permissible limit set by the HEC for the MS/MPhil degree thesis. I have also found the thesis in a format recognized by the BU for the MS/MPhil thesis.

Principal Supervisor's Signature: \_\_\_\_\_

Date: **21-07-2020** Name: **Dr. Tamim Ahmed Khan**

# Certificate of Originality

This is certify that the intellectual contents of the thesis **Model-based Testing for Ethereum Smart Contracts** are the product of my own research work except, as cited property and accurately in the acknowledgements and references, the material taken from such sources as research journals, books, internet, etc. solely to support, elaborate, compare and extend the earlier work. Further, this work has not been submitted by me previously for any degree, nor it shall be submitted by me in the future for obtaining any degree from this University, or any other university or institution. The incorrectness of this information, if proved at any stage, shall authorities the University to cancel my degree.

Signature: \_\_\_\_\_ Date: \_\_\_\_\_

Name of the Research Student: **Noureen Jan**

## Abstract

*Ethereum blockchain is popular blockchain between developers and financial related organizations. Ethereum allows to transfer cryptocurrency between two users on Ethereum blockchain as Bitcoin blockchain allows. After the release of Bitcoin Blockchain the finance related organizations used it and the demand more features in it because Bitcoin blockchain only allows you to transfer cryptocurrency. After seeing the increase interest of financial companies Ethereum blockchain release with more features, smart contract is one of them. Smart contract on Ethereum blockchain is an independent identity. Once smart contract will be uploaded on blockchain it will never be changeable. If any error, bug or failure arise after successful deployment you have to write a new smart contract which can cost you extra money which you can pay on deployment time. Smart contracts are written in high level programming language Solidity. Smart contracts are self-executable but they will run when an external application or system with inputs can call their any function. To overcome the issue of smart contract failure, we propose a model-based testing for Ethereum smart contracts. We extract model-based information to identify various aspects of smart-contracts to develop test cases and we then consider code-based information to execute test cases. We also provide test coverage criteria considering for smart contract interaction we write test cases for our system under test (SUT). It is pertinent to note that our proposed technique is equally useful to testing smart contracts in distributed systems in general.*

# **Dedication**

*This Thesis is dedicated to my beloved parents, respected teachers and all those who prayed for my success.*

## **Acknowledgments**

*I bestow all praises and appreciation to Almighty Allah, the most Merciful, Who gave me the understanding, courage and patience to complete this research.*

*I wish to express my deep sense of gratitude to my supervisor, Dr. Tamim Ahmed Khan for his able guidance and useful suggestions, which helped me in completing my thesis work. I gratefully acknowledge the generous supervision Dr. Tamim Ahmed Khan and for the care with which, he guided me and for their obliging and sympathetic attitudes.*

*Finally, yet importantly, I would like to express my heartfelt thanks to my beloved parents for their blessings, support and wishes for the successful completion of my thesis work.*

# Table of Contents

Approval Sheet.....	ii
Certificate of Originality.....	iii
Abstract .....	iv
Dedication.....	5
Acknowledgments.....	6
Table of Contents .....	7
List of Figures .....	10
List of Tables .....	11
Chapter 1 .....	12
Introduction .....	12
<b>1.1. Motivation</b> .....	14
<b>1.2. Problem Statement</b> .....	14
<b>1.3. Main Research Questions</b> .....	15
<b>1.4. Aims and Objectives</b> .....	15
<b>1.5. Our Contribution</b> .....	15
<b>1.6. Organization of Dissertation</b> .....	15
Chapter 2.....	16
Literature Review .....	16
<b>2.1. Testing</b> .....	16
<b>2.2. Smart Contracts</b> .....	16
<b>2.3. Smart Contracts Testing</b> .....	16
<b>2.4. Running Application</b> .....	18
<b>2.5. Blockchain</b> .....	19
<b>2.5.1. Types of Block chain</b> .....	22
<b>2.6. The Ethereum Block chain</b> .....	23
<b>2.6.1. Accounts</b> .....	24
<b>2.6.2. Transactions</b> .....	25
<b>2.6.3. Ethereum Virtual Machine</b> .....	25
<b>2.6.4. Ether and Gas</b> .....	26
<b>2.6.5. Blocks</b> .....	26
<b>2.6.6. Mining</b> .....	26

2.7.	<b>Smart Contracts</b> .....	28
2.7.1.	<b>Proxy Pattern</b> .....	29
2.8.	<b>Lifecycle</b> .....	29
2.9.	<b>DAO</b> .....	31
2.10.	<b>DApp</b> .....	31
2.11.	<b>Transaction Flow of Smart Contract based Applications</b> .....	31
2.12.	<b>Programming Languages</b> .....	33
2.12.1.	<b>Solidity</b> .....	33
2.12.2.	<b>Node.js</b> .....	33
2.13.	<b>Tools</b> .....	33
2.13.1.	<b>Ganache</b> .....	34
2.13.2.	<b>Truffle</b> .....	34
Chapter 3	.....	35
Proposed Methodology	.....	35
3.1.	<b>Research Methodology</b> .....	35
3.1.1.	<b>Literature Review</b> .....	35
3.1.2.	<b>Model Proposal</b> .....	36
3.1.3.	<b>Validation</b> .....	36
3.2.	<b>Our Approach</b> .....	36
3.2.1.	<b>Smart Contracts structure</b> .....	37
3.2.2.	<b>Layered Architecture of Blockchain</b> .....	37
3.2.3.	<b>Ethereum Blockchain</b> .....	38
3.2.4.	<b>Testing Technique</b> .....	39
3.2.5.	<b>Coverage Criteria</b> .....	43
3.3.	<b>Running Applications</b> .....	43
3.3.1.	<b>Case Study 1:</b> .....	43
3.3.2.	<b>Case study 2</b> .....	63
Chapter 4	.....	65
Results and Evaluation	.....	65
4.1.	<b>Testing Results</b> .....	65
4.1.1.	<b>Case Study 1</b> .....	65
4.1.2.	<b>Case Study 2</b> .....	65
4.2.	<b>Test Cases Mapping</b> .....	66



<b>4.3. Comparison of techniques</b> .....	68
<b>4.4. Generalization of our proposal</b> .....	69
<b>4.5. Validation</b> .....	70
Conclusion and Future Work .....	72
References .....	74
Appendix A.....	80

## List of Figures

<b>Figure 1:</b> A basic structure of Smart Contract [3].....	12
<b>Figure 2:</b> Blockchain Architecture [38].....	13
<b>Figure 3:</b> Ethereum Blockchain Layers [14].....	14
<b>Figure 4:</b> Smart Contract Example.....	19
<b>Figure 5:</b> Blockchain Generation [22] .....	19
<b>Figure 6:</b> Chain of Blocks [36] .....	20
<b>Figure 7:</b> Permission less Vs Permissioned types of Blockchain [46] .....	22
<b>Figure 8:</b> Ethereum Transaction [49] .....	24
<b>Figure 9:</b> Schematic of a blockchain platform with smart contracts [62].....	28
<b>Figure 10:</b> Smart contracts Proxy Pattern [63] .....	29
<b>Figure 11:</b> Decentralized Applications Structure [53] .....	31
<b>Figure 12:</b> Ethereum Network Node's View .....	32
<b>Figure 13:</b> Transaction Flow of Smart Contract .....	32
<b>Figure 14:</b> Node.js Version Command.....	33
<b>Figure 15:</b> Initial View of Ganache Workspace .....	34
<b>Figure 16:</b> Truffle Version Command .....	34
<b>Figure 17:</b> Research Methodology.....	35
<b>Figure 18:</b> A basic structure of Smart Contract [3].....	37
<b>Figure 19:</b> Blockchain Architecture [38] .....	38
<b>Figure 20:</b> Consumer Role Code .....	46
<b>Figure 21:</b> Distributor Role Code .....	46
<b>Figure 22:</b> Farmer Role Code .....	47
<b>Figure 23:</b> Retailer Role Code .....	47
<b>Figure 24:</b> Ownable Code .....	48
<b>Figure 25:</b> Supply Chain Code .....	48
<b>Figure 26:</b> Sequence Diagram .....	51
<b>Figure 27:</b> Fault Seeding in Smart contract.....	70

# List of Tables

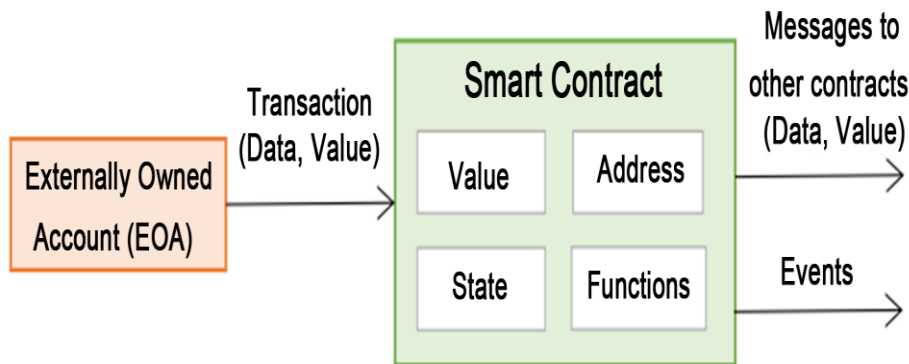
<b>Table 1:</b> Pre-conditions .....	39
<b>Table 2:</b> Test cases related to Pre-conditions .....	40
<b>Table 3:</b> Post-conditions .....	40
<b>Table 4:</b> Test cases related to Post-conditions .....	41
<b>Table 5:</b> Contract Invariants Classes .....	41
<b>Table 6:</b> Test cases related to Contract Invariants Classes .....	42
<b>Table 7:</b> Roles .....	42
<b>Table 8:</b> Test cases related to Roles .....	42
<b>Table 9:</b> Test Cases Mapping.....	66

## Chapter 1

# Introduction

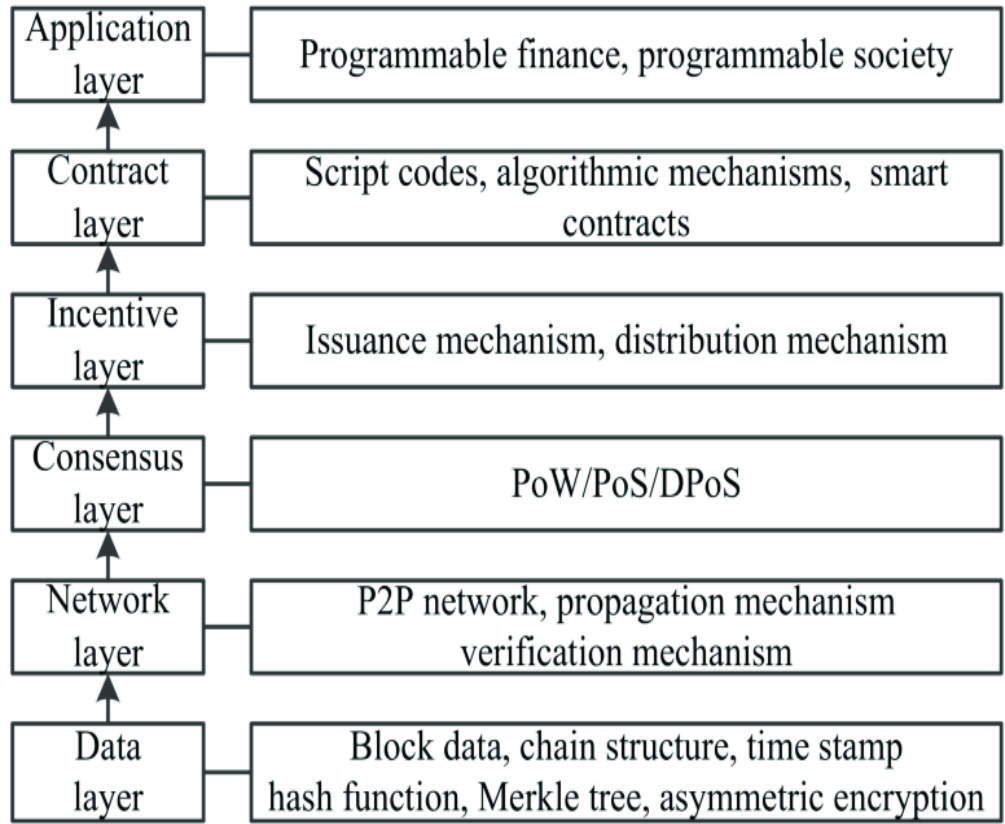
Smart contracts get automatically executed and perform processing frequently in an IT-environment without any human involvement [1]. Smart contracts are written in high level programming languages and are executed on blockchain environment [2]. After the deployment of smart-contract, blockchain will automatically verify, execute and enforce the contract terms between both parties i.e., service provider and consumer. These contracts are called smart contracts because they can be partially or fully self-executing and self-enforcing [2].

Smart contracts concept was first proposed in 1994 by Nick Szabo [3]. Smart contract is a small piece of code with unique address that resides on blockchain [4]. A smart contract contains variables and set of executable functions. Whenever a transaction is executed it contains parameters required for execution of function. On execution of a function, variables state will be changed in smart contract on the basis of logic implemented in function. The structure of smart contract is showed in Figure 1.



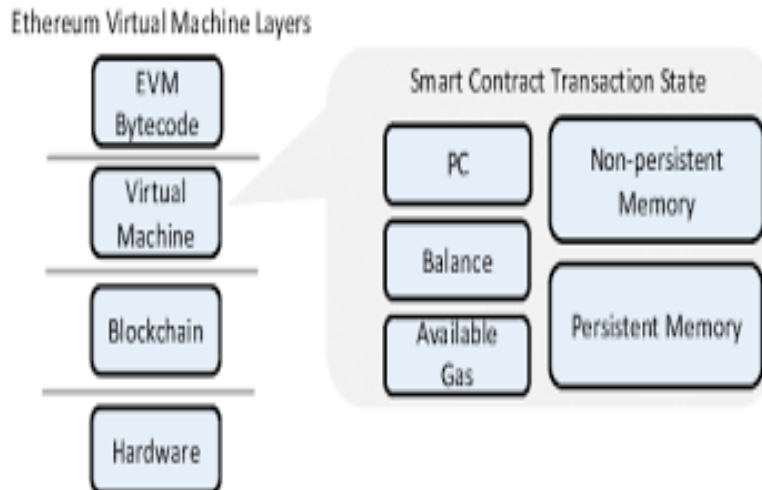
**Figure 1:** A basic structure of Smart Contract [3]

The blockchain architecture is basically divided into six layers (Figure 2): the data layer, the network layer, the consensus layer, the contract layer, the service layer, and the application layer. The data layer and network layer are the lower levels. These layers generate, validate, and store the data and information. The consensus and contract layers are the intermediary between the lower and upper levels. The consensus layer is mainly consists of PoW, PoS, DPoS, and PBFT. The contract layer includes smart contract, consensus protocol, and incentive mechanism. The upper level is at the top of the architecture, including the service and the application platform [38].



**Figure 2:** Blockchain Architecture [38]

We can write smart contracts in high level programming languages like Python and Solidity. In blockchain architecture, on contract layer our smart contracts work. We can write smart contracts on Ethereum blockchain. On Ethereum blockchain, a smart contract is a file of written code which enforces and defines agreements between Ethereum users. On Ethereum smart contracts are usually written in solidity language and are then compiled in to bytecode for deploying on Ethereum Blockchain. On every node Ethereum virtual machine (EVM) is running and executing smart contracts byte code. Almost 70 different opcodes includes in EVM bytecode for computation and communication with the underling blockchain [4]. Every online smart contract EVM bytecode is publically accessible on the blockchain. All deployed contracts accessed three things; own account balance, its byte code and private storage. EVM byte code generally access two types (persistent and non-persistent) of private storage. In persistent storage a key-value store continued through transactions. While in non-persistent storage discards it's content once transaction is completed. A gas price is charged on every EVM instruction [4]. Figure 3 shows how a smart contract fits in to Ethereum [14].



**Figure 3:** Ethereum Blockchain Layers [14]

### 1.1. Motivation

Our business logic is written in smart contract in the form of computer code, Smart contract is an important component of all those applications which contracts deployed on blockchain. Smart contract cannot be changed after deployment and if any error or bug is identified by any user after it then all other users will must be effected by it too.

We require testing technique for Ethereum smart contracts which considers aspects such as pre-conditions, post-conditions, invariants etc. since a contract has a Pre-condition and a Post-condition [65]. This must be a technique assisted by model-based information with test case development technique considering contract implementation. We limit our discussion to Ethereum blockchain smart contracts and then we generalize our technique for distributed system with similar base.

### 1.2. Problem Statement

There are few techniques that are used to test runtime behavior of smart contract with application and response accordingly. There are also limited techniques used to test smart contract but there is no testing technique for smart contracts that considers pre and Post-conditions. We require a model-based testing for smart contracts which is accompanied by model-based coverage criteria and considers essential constructs of smart contracts.

### **1.3. Main Research Questions**

Q1. How can we test smart contracts using a model-based technique?

Q2. How can we test smart contracts by providing a model-based coverage criteria?

### **1.4. Aims and Objectives**

Objective of this research is to propose a model-based testing for smart contracts. The proposed model will provide:

- Devise a model-based test case aspects and development technique for smart contracts
- provide a model-based coverage criterion for testing smart contracts

### **1.5. Our Contribution**

In our research we proposed a model-based testing for Ethereum based contracts and we supplemented it with a general smart-contract testing technique. As smart contracts are an independent entity on Ethereum network of any external system wants to communicate with the smart contract any function then it must have the address of the contract. After calling the specific function of the smart contract by using its address, smart contract will be triggered. We defined a coverage-criteria for our testing. We consider both positive and negative intent test cases for our system under test (SUT). We designed a matrix for our test cases having a pre-condition, a Post-condition, expected output, actual output and result.

### **1.6. Organization of Dissertation**

The rest of the thesis is organized as follows. Chapter 2 is contains the background of smart contracts and its testing. Chapter 3 contains complete technical background of Ethereum and its smart contracts. Chapter 4 is on methodology of our research. Chapter 5 is on Conclusion.

## Chapter 2

# Literature Review

This chapter introduces previous research studies of smart contracts testing and the background of smart contracts on blockchain. Also we have introduced our running example of system which we have taken for testing.

### 2.1. Testing

Testing is an important technique to validate the system under test (SUT) in information and communication technology (ICT) systems [5]. Test case is a set of different conditions and inputs. Test cases are written to validate the system under test (SUT) outcomes with the desired outcomes [5].

Model-based testing is defined as the more efficient testing in which we test cases generated on the basis of defined model using the combination of system under test requirements and specific functionality.

### 2.2. Smart Contracts

Smart Contracts are a well-studied areas, a lot of research work has been done in the area. Smart contracts concept presented by Nick Szabo in 1994. He defined it as “a computerized transaction protocol that executes the terms of a contract” [6]. Smart contracts structure makes them self-enforceable in order to diminish the need for trusted mediators between transacting parties [7]. For the development of smart contracts developers mostly prefer Ethereum platform. The main components of smart contract required for transactions are based on functions and state machine. The design and the implementation of the Ethereum are totally independently from the cryptocurrency Bitcoin. A high-level programming language called Solidity is used to write smart contracts and decentralized applications (Dapps). The programmer can create their transaction state, formats, events, functions, and rules for ownership. A special virtual machine called Ethereum virtual machine (EVM) designed for the execution of smart contracts [8].

### 2.3. Smart Contracts Testing

Mense et al. identify the known security vulnerabilities of smart contracts and provide an updated in-depth analysis of existing smart contract vulnerabilities. Investigate the security code analysis



tools used to identify vulnerabilities and bugs in smart contracts [9]. Visual contracts are defined as the visual representation of smart contracts. Tamim et al. proposed a model-based coverage for helping the tester to implement test cases by automating the decision, if the response from the operation being tested is correct. This information is present in visual contracts and should be reused rather than re-implemented [10]. The use of graph transformation systems specifying service interfaces for the derivation of model-based coverage criteria [11]. Tamim et al. proposed a model to generate test cases from visual contract. From visual contracts direct graphs (DG) extracted to originate test cases and if all these test cases executed on system under test (SUT) then it will be guaranteed to accomplish full coverage [12]. A technique based on combining static and dynamic analysis process model-based coverage proposed. Ardit et al. consider all the security vulnerabilities of smart contracts which were already known. Also proposed an in-depth updated analysis of all the vulnerabilities in the prevailing smart contracts and tools for code analysis. The purpose of these tools were to detect bugs and vulnerabilities in smart contracts [13]. Wang et al. analyzed nondeterminisms in the smart contract execution that cause unpredictable payments and potential financial loss [14]. Zixin et al. proposed a new tool MuSC for mutation testing of smart contracts. MuSc tool is for Ethereum based smart contracts, it contains all the operators used in Ethereum based smart contract programming language. It facilitate developers to do mutation testing on user-defined test net [15]. Solidity is a programming language which is used for developing smart contracts in Ethereum. W.K Chan and Bo Jiang proposed a fuzz testing service named fuse for the testing of Ethereum based smart contracts [16]. William et al. proposed an optimization based technique integrated with Mythril-Classic. This technique allow to detect depth-n vulnerabilities by analyzing data dependency [17]. Parizi et al. considered open source tools for automatic detection of security vulnerabilities in Ethereum based smart contracts written in solidity and analyzed all these tools by doing 10 real-world smart contracts testing on them. The end result of research was that the SmartCheck tool is more efficient and accurate than all other tools to find security vulnerabilities in Ethereum based smart contracts [18]. Fu et al. proposed an automated framework named EVMFuzz to find the security vulnerabilities in Ethereum Virtual Machines (EVMs) developed in different programming languages [19]. Haya and Khaled proposed a blockchain based solution for the detection of deepfake video and audio. It helps artists to find that the requested video or digital content is traceable or not. In proposed solution, a smart contract developed to facilitate secondary artists to take permission from original artists to copy or

edit digital content or video [20]. Destefanis et al. considered the case study of Parity smart contract library in which due to a delay in library bug fix activity, a non-experienced developer did a hacking activity due to which 513774.16 ethers frozen. By considering this case study they clearly stated that Blockchain oriented software engineering is required to avoid such consequences again [21]. Valentin and Maria introduced a new greybox fuzzer testing technique for smart contracts named HARVEY [72]. They addressed two key challenges that faced during real world contracts fuzzing are random input mutations and state space exploration. HARVEY [72] tested by focusing on these two challenges and is used in different industry companies that are concerned majorly with security. Piter and Richard introduced a new set of operators on the basis of previous work and test these operators on scale and for further improvement of mutation score they introduced a condition based on smart contracts gas limits [73]. Andesta et al. proposed different mutation operators for the testing of smart contracts written in Solidity language [74]. These mutation operators evaluated on fixed buggy smart contracts and find the percentage of bugs generated by introduced operators. For the generation of mutants for smart contracts they used Universal Mutator with their defined mutation rules [74]. Ashraf et al. introduced a GasFuzzer that deals with the transaction gas allowance to exploit security vulnerabilities during gas allowance and consumption dimensions [75]. Patrick and Lin et al. introduced a new tool named Deviant for the mutation testing of smart contracts written in Solidity [76]. Deviant [76] automatically generated mutants for the solidity project to evaluate different faults that may occur during programming process. Akca et al. proposed an automated technique to evaluate the vulnerabilities of smart contracts on code execution and also provides detection of some vulnerabilities types that are not analysed much in previous works [77]. Nehai et al. proposed a method to check model of the application based on smart contracts [78]. The Kernel layer, application layer and the environment layer are the three-fold modelling process on the basis of which a model is build [78]. Translation rules from Solidity to NuSMV [78] language have been provided to build the application layer.

#### **2.4. Running Application**

We take a running smart contract based supply management application named “Parmigiano Reggiano Supply Chain Management Application” from Github. We used this application smart contracts in our research for testing. It is developed to manage cheese sale and keep track of whole sale from farmer to the buyer. This application contains role based contracts and a main contract

to handle whole sale. Farmer, distributor, retailer and customer are different actors in it and have different roles. Smart contracts written in solidity, we use Git, Ganache and Truffle for testing. Python and Node.js are prerequisite for running Ganache and truffle. Some events shown in Figure 4 written in smart contract.

```

Contract Ownable
{
    ...
    event ShippedByFarmer(uint upc);
    event ReceivedByDistributor(uint upc);
    event ProcessedByDistributor(uint upc);
    event PackagedByDistributor(uint upc);
    event ForSaleByDistributor(uint upc);
    event PurchasedByRetailer(uint upc);
    event ShippedByDistributor(uint upc);
    ...
}

```

**Figure 4:** Smart Contract Example

## 2.5. Blockchain

Blockchain concept was first born in 2008 when bitcoin was introduced. An individual or group name under Satoshi Nakamoto published a paper titled: "Bitcoin: A Peer-to-Peer Electronic Cash System". In January 2009, when an open source program is introduced bitcoin has grown in popularity because any one can install this program and can join the Bitcoin peer-to-peer network. "Block" and "chain" two separately nouns were used in Satoshi Nakamoto (2008) while in 2014 the concept merged in to a word "Blockchain".

Three generations of blockchain proposed [22, 23 and 24]

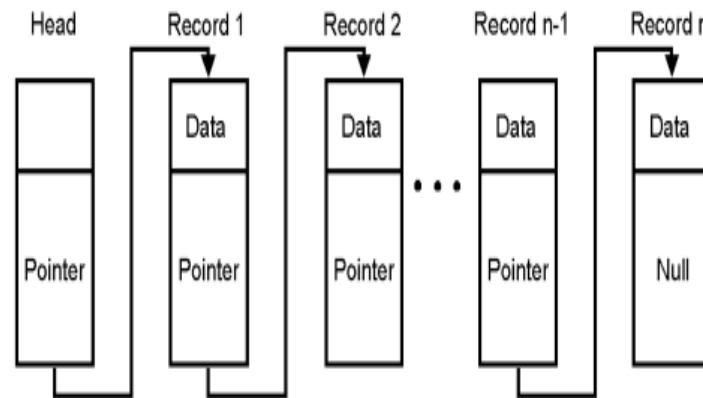
1. Blockchain 1.0 (for cryptocurrency)
2. Blockchain 2.0 ( for financial services)
3. Blockchain 3.0 (for the authentication of digital identity, currency economics)

Type	Description	Examples
Blockchain 1.0	Currency	Cryptocurrencies like Bitcoin. Was first introduced in 2009.
Blockchain 2.0	Contracts	Financial services, crowdfunding, Bitcoin prediction markets, smart property, smart contracts. Was introduced through the release of NXT in 2013.
Blockchain 3.0	Justice, efficiency and coordination applications beyond currency, economics, and markets	Digital Identity, Intellectual Property Protection, Governance Services, Elections. Solutions within these areas of applications are starting to take form.

**Figure 5:** Blockchain Generation [22]

Swan [22] presented a short-term assessment of three different generations of blockchain as shown in Figure 5. Blockchain used in different sectors context and consequently addressed in different disciplines, we can say that this is a one reason why we don't have any general accepted definition between practitioners and researchers of blockchain [25]. The second reason is that Blockchain is an emergent research [26, 27] that lacks cohesive terminology and well defined concepts that characterize Blockchain.

Blockchain have no specific definition, different authors define it in different ways. Olnes [34] defined blockchain as an open, distributed, and trust less database on the Internet. Other researchers such as [28, 29–34] agrees that Blockchain is a distributed data structure, database or system. Blockchain is a sequence of blocks containing a complete list of transaction records [35]. A block contains previous block hash in the header, every block has only one parent block as shown in Figure 6. It is worth noting that uncle blocks (children of the block's ancestors) hashes would also be stored in blockchain [36]. The first block of a blockchain have no parent block and is called genesis block. Blocks are created by network participants who are processing transactions by using client software, it is impossible in the mid to delete or insert a new block because hashes will never match. Bitcoin hashing scheme proof of work is similar to Hashcash and based on SHA-256 hash function [37]. In Bitcoin transaction, blocks contains transactions which are hashed with Markel Tree [39], [40]. Markel tree is one of the type of binary tree having root nodes and leaf nodes. Root node is the hash of its all child nodes [37].



**Figure 6:** Chain of Blocks [36]

Blockchain has following key characteristics based on the survey did by Zhang et al. are as following:

## **A. Decentralized**

In traditional centralized transaction systems there must be a central authority which validates all the transactions before execution. These centralized transaction systems cause a bottleneck on server by effecting its performance. Blockchain is a decentralized system which means there is no central authority required in it to validate the transactions. Transactions on blockchain network can be accompanied between two users in a peer to peer manner in which no central authority is required to validate the transactions. This characteristic of blockchain improves its performance because all the load on the server side in centralized transaction systems is now ended in decentralized transaction systems.

## **B. Persistence**

In blockchain network, transaction dispersion over the network can be confirmed and recorded in blocks distributed on whole network. Each broadcasted block would be validated by other nodes and transactions would be checked. Any type of misrepresentation could be identified easily.

## **C. Anonymity**

All users on network can be interacted with blockchain network by any generated address. Blockchain network is a decentralized network so there is no central authority to take all the private information of users to validate. Users can generate multiple addresses to interact with blockchain network to avoid their identity exposure. All the transactions on the network have some amount of security in it because of this mechanism.

## **D. Auditability**

All transactions on the blockchain network can be validated and logged with timestamp, users on the network can easily trace or verify the previous transactions record by accessing any node over the distributed network. In bitcoin blockchain previous transaction can be traceable iteratively. This will improves the traceability and transparency of the transactions related data which is stored on blockchain network.

### 2.5.1. Types of Block chain

Blockchain networks can be designed on the bases of different deign or model options. The Data on blockchain network can be observed, all design and model options for blockchain networks can be classified on the bases of who should be allowed to participate on the network [41].

Based on that there are essentially three types of blockchain are introduced by some researchers. Wust et al. introduced two blockchain types permissioned and permissionless blockchain in his research [42]. Vokerla et al. introduced three types of blockchain in his research which are: public, private and consortium blockchain [43]. Bitcoin and Ethereum are the examples of permissionless blockchain. Public blockchain are permission-less blockchain and is defined as the blockchain which is decentralized and open, anyone can access it without any permission [42]. Private Blockchain are permissioned blockchain and is defined as the blockchain which have a central authority to validate or give permission to a single entity to participate or read the blockchain [42]. Consortium blockchain are semi-decentralized in which more than one company is taking part in its operations [43].

	Permissionless	Permissioned
General purpose	Ethereum	Monax's eris-db
Specialised	Bitcoin	Multichain

**Figure 7:** Permission less Vs Permissioned types of Blockchain [46]

Although there is no categorization related to blockchain design types or models. However, some of the researchers tried to fill this gap by proposing a comparison between different blockchain types such as a researcher work shown in Figure 7 which gives a comparison between permission less/permissioned and general purpose/specialized blockchain.

#### 2.5.1.1. *Ethereum*

Ethereum is a platform which is based on distributed computing. Ethereum practices a blockchain for saving the state of user's accounts, program codes and its associated states. It also allows to develop smart contracts which will be discussed in further section 3.2. Some scripting languages

are supported by Ethereum to write smart contracts which will be then compiled in to byte code that is executed on Ethereum Virtual Machine (EVM) [60, 61].

#### **2.5.1.2. *Monax's eris-db***

Monax's eris-db was a first permissionless blockchain based designed blockchain client free system for developers to develop or blockchain applications and smart contracts based blockchain applications related to business processes and systems [46].

#### **2.5.1.3. *Bitcoin***

Bitcoin was the first decentralized cryptocurrency introduced in 2008 with blockchain concept. After the release of Bitcoin, Finance industry show more interest in it. As a result Bitcoin had a 60 billion dollars market capitalization which was the highest among all other cryptocurrencies. Bitcoin blockchain introduced Proof-of-Work consensus mechanism to solve the double expenses problem. To finalize the block in Bitcoin it takes almost 10 minutes.

#### **2.5.1.4. *Multichain***

Multichain is a platform which facilitate users to create and deploy private blockchains. Problems related to the openness, privacy and mining are solved by it with the integration of user permissions management in it [44, 45].

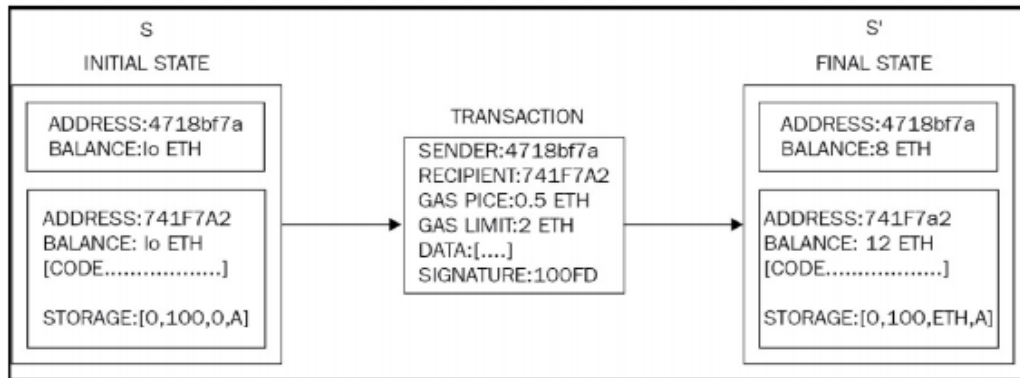
We are working on Ethereum blockchain which is a permissionless blockchain. In the next section we tried to cover all the related concepts of Ethereum Blockchain. We take Ethereum Blockchain for our thesis work because it allows developers to write custom smart contracts and it supports multiple scripting languages.

### **2.6. The Ethereum Block chain**

The first blockchain introduced with name Bitcoin and after the successful release of bitcoin the financial industry shows more interest in it. After using the bitcoin blockchain the financial industry demands more features in blockchain system because the bitcoin blockchain only provides functionality of transfer electronic money peer to peer [47]. When the demand of financial industry increases researchers think about another network which fulfils all their demand. In year 2014 a programmable blockchain released with name Ethereum which fulfils all the demands of financial industry [48]. Ethereum have different approaches other than electronic money transfer

between peers like users add their own operations and complexity [49]. Therefore we can say that for complex business logics Ethereum blockchain is a best option.

The design of Ethereum blockchain is similar to state machine [49, 50]. A transaction is created whenever a user wants to change the data on Ethereum blockchain. The transactions will be collected and processed incrementally each of which will elaborate how the data will change from current state to next state, as in Figure 8.



**Figure 8:** Ethereum Transaction [49]

Ethereum Classic (ETC) [50] and Ethereum (ET) [51] are two running Ethereum networks. Ethereum Classic is an old branch of Ethereum which is also working these days separately with its own cryptocurrency and community and Ethereum is a new branch of Ethereum network which is also working separately. In this thesis, we used Ethereum new branch for the deployment and testing of smart contracts. The popular cryptocurrency on Ethereum network is called “Ether”.

### 2.6.1. Accounts

In Ethereum, users must have an account to process transactions and to communicate with blockchain so we can say that accounts on blockchain plays an important role. External owned account and smart contracts account about which we can say it’s a contract account, these are two types of account on Ethereum. Both types of accounts are same but controlled differently. The External Owned Accounts (EOA) are controlled by private keys while smart contracts are controlled by internal code which can be executed by EOA’s. Accounts have 20-byte address as they are treated as state objects which can be for external agent’s identification that can be contract account or EOA accounts [54]. All accounts have public key by using which they signed transactions. The characteristics of account in blockchain can be classified in to four categories which are as following.



### **A. Nonce**

It's a counter that is used to count the number of transactions executed from an EOA account or number of contracts created from a contract account.

### **B. Current Balance**

Current Balance amount is in Ethers which an account have.

### **C. Code**

A contract account have an external code which can be executed by EOA's.

### **D. Storage**

Contract account on blockchain have a permanent storage to save data.

## **2.6.2. Transactions**

Ethereum is an account based model as we have banking system user must have an account on Ethereum to do transactions or any operation they want to do on blockchain. Whenever a transaction information or value is executed between two users account the state of Ethereum changes. On Ethereum transactions are of three types transfer Ether, create a contract and a call to contract.

In Ethereum a smart contract cannot execute a transaction, a smart contract can call internally other account smart contract for transaction, and this internal transaction call can be called as a message call. When a message call can be held it contains the calling function which can be activated in called contract.

## **2.6.3. Ethereum Virtual Machine**

Ethereum virtual machine is a computing machine which can be used for the execution of EVM code. EVM code is a byte code. EVM can be used for the validation of transaction details like signature in the transaction, correct number of values and matching of nonce with the specific transaction account nonce. It can also be used for checking the gas is enough or not to execute the transaction. EVM can also transfer ethers from one account to the particular account. EVM also calculate the gas and transaction fee of a transaction to initiate miner's gas payment.

#### **2.6.4. Ether and Gas**

Ether is the crypto currency used by the Ethereum blockchain. It contain a specific value like other currencies have. Ether can be used for transactions and for paying gas fee which can be calculated on transaction computation to the miners.

Computation required on using resources of blockchain and for transactions blockchain charged some fee which is called gas. The price of gas is dynamically changed with the change of Ether price. Gas is calculated on the bases of computation required multiplied with the current gas price.

#### **2.6.5. Blocks**

Block is an essential component of Ethereum blockchain. On blockchain blocks contain information related to transactions. Block contains all mined transactions set and a block header. The chain of blocks can be created by hashing the block. Hash of the block saved in next block for reference and by doing this with all blocks a chain is created on blockchain.

The Ethereum Blockchain starts it life with a zero block called genesis block. After every 14 seconds Ethereum blockchain refresh its state because the difference between blocks in almost 14 seconds so we can say that it takes 14 seconds to refresh its state. A block on Ethereum contains hash of previous block and its own previous work, on the basis of these two things hash of the transaction will be calculated on Ethereum blockchain. The maximum block size on Ethereum blockchain is around 1,500,000 gas.

#### **2.6.6. Mining**

The process of creating new blocks and validating transactions on block is known as mining. Miners on blockchain get transaction from the transaction pool, run it on EVM and solve its nonce. After successfully solving the nonce of block they create a new block that fits in to the chain and add it in to the transaction pool. Miners awarded differently with Ethers on every task like on adding a new block in the chain they will be awarded with three Ethers.

A block consumes 15 seconds to create on Ethereum network which is equal to the punctuation between nodes on the Ethereum blockchain, this phenomena give guaranty that no malicious attack can arise to change the history.

### **2.6.6.1. Proof of Work (POW)**

Ethash is a POW algorithm used on Ethereum blockchain. This algorithm contains finding a dataset, a nonce and a header. To find these three, headers of all blocks on blockchain are hashed together to create a seed. Seed will be used to generate a cache known as pseudorandom cache. By using non-cryptographic hash function this cache generates a dataset. The dataset, nonce and header all these three things are repetitively hashed until the satisfied difficulty target [52].

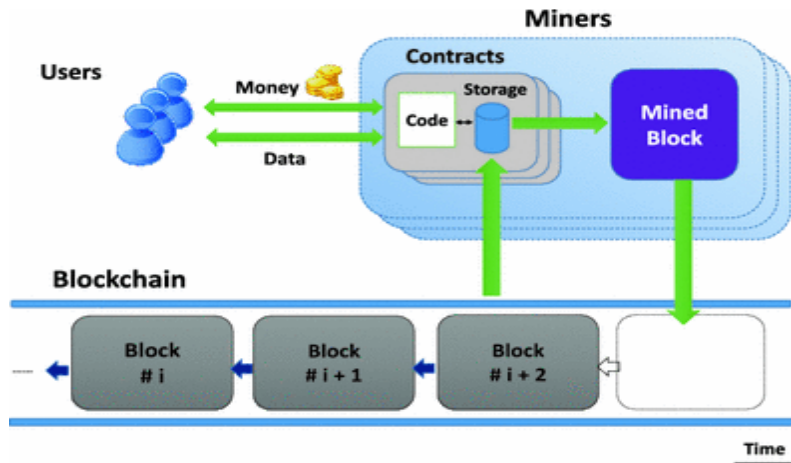
### **2.6.6.2. Proof of Stake (POS)**

The substitute method to influence consensus on blockchain is known as proof of stake (POS). A task is given to the validators to suggest transactions as blocks to the blockchain to reach consensus. To do it validators have to solve a POS algorithm to avoid overflow of suggestions on blockchain. The validator must have to add the solution with the suggested block for validation check, if the block is valid then it will be added to the transaction blocks pool [53].

Stakes are virtual resources that are required by validators to solve the POS algorithm. Discovering new blocks with solution speed is totally dependent on stakes [53].

### **2.6.6.3. Smart Contracts Execution**

In this section we explain the general concept of smart contracts execution. In general a smart contract is defined as the business logic of any decentralized application or an agreement between parties. The business logic of the application is written in code and deployed on blockchain by which application will be interact. Functions of the smart contracts will be called by application after that smart contract function will be triggered. As Delmolino et al. represent the architecture of blockchain system with smart contracts and also represent the process of sending money by users as shown in Figure 9 [62]. Smart contracts is the most important component of any DApp based cryptocurrency system. In the next section we can explain the smart contracts execution on Ethereum blockchain.



**Figure 9:** Schematic of a blockchain platform with smart contracts [62]

#### 2.6.6.4. *Smart Contract Execution on Ethereum*

In Ethereum, after the deployment of smart contracts it contains specific address on network. To run a smart contract on network user required an account on Ethereum network with balance amount in Ethers. When a transaction is initiated a gas amount is paid to miners by the sender other than the amount which is being transferred in transaction. If the transaction is invalid than the gas amount will be paid to the miners who validate transaction on the network and the remaining amount will be refunded to the sender account. If the transaction is valid than the gas amount is given to miners. After that the smart contract will be active and the explicit function of smart contract will be called for operation on transaction.

### 2.7. **Smart Contracts**

So far we have mentioned a small introduction of smart contract and Ethereum based smart contracts in chapter 1. But in this section we will give a whole overview of smart contract in relation to the Ethereum.

Smart contracts are not more than a piece of code which mainly contains the business logic of the application. Smart contracts run on top of the blockchain network and more like the other classes concept in object oriented programming. Smart contracts are exist on Ethereum network independently. Smart contracts on Ethereum can be executed or trigged whenever any function of the deployed contract by either transactions or messages. Smart contracts byte codes are stored on blockchain. The bytecode contains all the rules and agreements between two parties without the involvement of any third party. As we know smart contracts are consist of different functions then

we can say that these functions are must be in the byte code of contract. To keep the database clean and save the space Ethereum allows smart contracts to automatically self-destroy when they are not doing any operations or saving values. A smart contract represented by 20-byte address like EOA accounts have address for identification.

Smart contract can be written in high level programming language like solidity for Ethereum network. When a smart contract is deployed on the network, EVM will compile the code in to bytecode. The smart contracts on Ethereum make it more flexible for different applications. Ethereum allow a developer to write any type of smart contract.

### 2.7.1. Proxy Pattern

The biggest disadvantage of writing smart contracts using traditional pattern is that once smart contract will be deployed on blockchain than it will never be changeable. As we know developers doing changes continuously for bug fixing but this feature is not supported by Ethereum environment. To overcome this problem proxy pattern will be used in writing smart contracts. The function of proxy contract in it is to redirect the contracts delegate calls to the logical contracts as shown in Figure 10. Separate data and logic contracts and spate data and logic contracts with key-value pair are the two approaches used in it [63]. ZapplinOs provides the feature of developing upgradeable smart contracts [64].

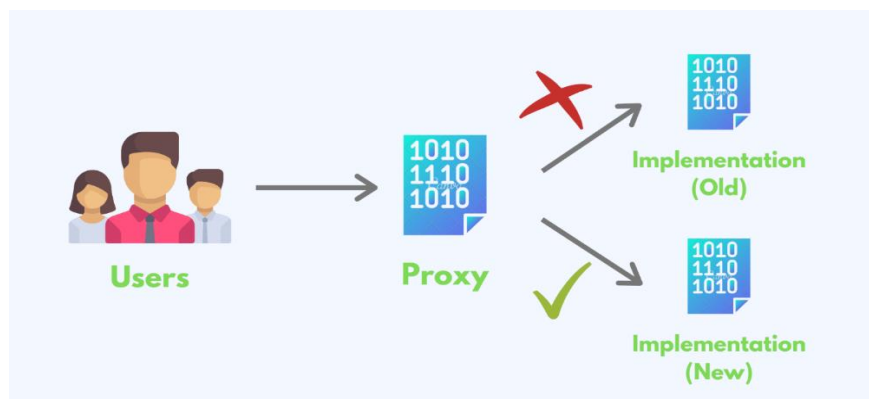


Figure 10: Smart contracts Proxy Pattern [63]

## 2.8. Lifecycle

As we know that the smart contract is an independent identity on the blockchain, once the code of the smart contract deployed on blockchain it will never b modifiable after deployment. If there is

any bug or error in deployed smart contract you have to write separately a new smart contract and then deploy it. Smart contract is an independent identity, there is no strong link between application and smart contracts. If the calling application will be destroyed there is no effect of application on smart contract.

When a smart contract is submitted to deploy on blockchain, it will be exposed to miner nodes to validate the execution of smart contract. A gas fee will be given to miners as they used computational power to validate the contract. After this point smart contract will be public to all the users on blockchain. Any one on the blockchain can access the smart contract.

After the deployment of smart contract if any user want to interact with the smart contract then they must have some inputs for the calling function. After execution of it smart contract state will be changed and a transaction will be created on the blockchain. All these elements will be submitted to Ethereum ledger and will be validated by consensus mechanism [55]. After validation an amount will be transferred to the respective account and the contract agreement fulfilled. It's an infinite amount that can be created on the network. We also have 'templated' on Ethereum network to create smart contract with main functions to achieve similar behaviour. These are tokens that can be defined as common list of rules and are called ERC (Ethereum Request Comments). Some of these are described below [54]:

**ERC20:**

It is the most commonly used standard on Ethereum because it provides a simple interface. This standard is used to build tokens and can be reused in any application. In decentralized exchange platform for the exchange of wallet ERC-20 standard used [54].

**ERC223:**

If a token is written in ERC20 standard and sends a request to the contract which is not written in ERC20 standard, this token will not be accessible again. ERC223 introduced to validate the request standards and to avoid this issue [54].

**ERC721:**

These standards developed a complete unique tokens. We can use this standard for the representation of assets that cannot be duplicated [54].

As mentioned in earlier sections that DApp have backend and frontend. Backend development is done in Ethereum and Solidity so we can say that smart contracts are the backend system of DApp. DApp are software applications that can be communicate with the smart contracts deployed on blockchain. This is the way by using which users can interact with the smart contracts. At the end we are able to say that the DApp is an interface for the smart contracts.

## 2.9. DAO

The collection of smart contracts or the type of contracts on blockchain is known as decentralized autonomous organization (DAO). It is coded for automate the services of organization but the main purpose of it is to be automatically driven by smart contracts [53].

## 2.10. DApp

Decentralized Applications can automate business logic of application by using smart contracts. DApp also have backend and frontend like other applications have. The frontend of the DApp is developed by combining Html, CSS and JavaScript. The backend of DApp can be developed using Ethereum and Solidity as shown in Figure 11. This will allow web applications to be moderately decentralized [53].

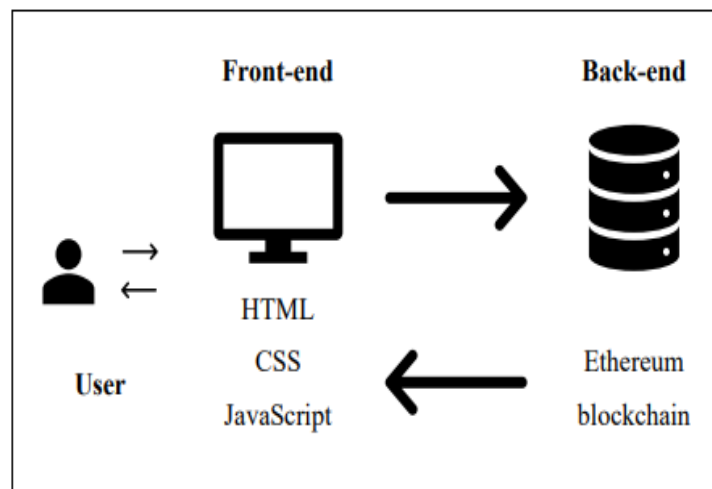
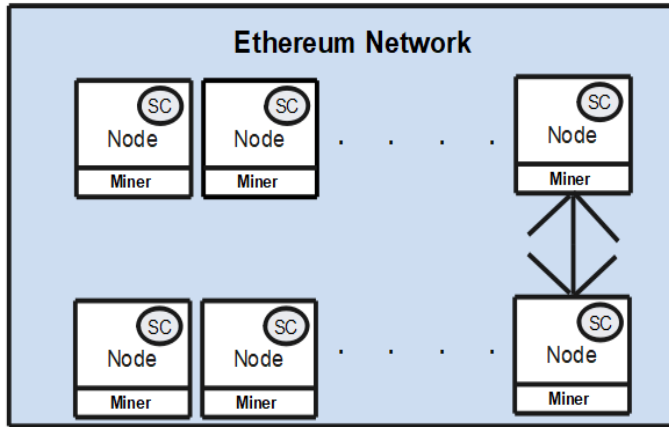


Figure 11: Decentralized Applications Structure [53]

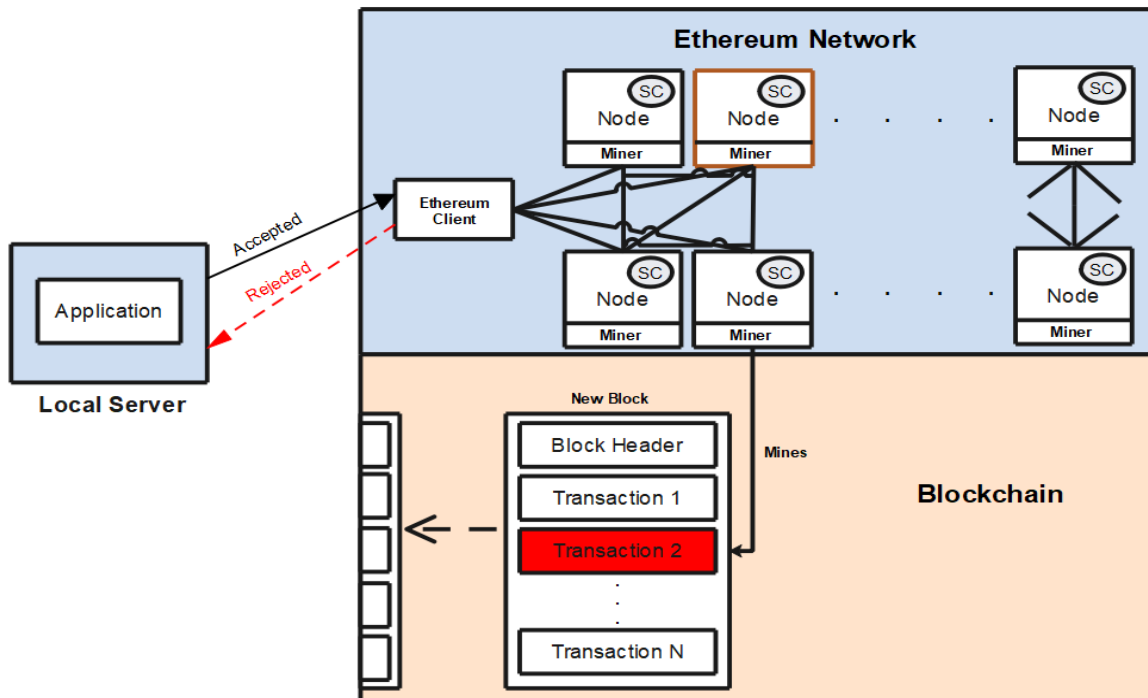
## 2.11. Transaction Flow of Smart Contract based Applications

Once we write a smart contract and deploy it on Ethereum network. Deployment mechanism ensure that all nodes on the network have the same copy of smart contract on their nodes as shown in Figure 12.



**Figure 12:** Ethereum Network Node's View

Whenever a transaction is generated from the application it will be transmitted to the Ethereum client. Ethereum client will send the response to the application after processing transaction. Ethereum client will redirect the transaction to nodes and all nodes will start validating it. Here nodes validating mean different miners will start to mine the transaction block to get reward. Once a block is mined by any miner it will be added to the blockchain where multiple blocks merge to make a transaction as shown in Figure 13.



**Figure 13:** Transaction Flow of Smart Contract



## 2.12. Programming Languages

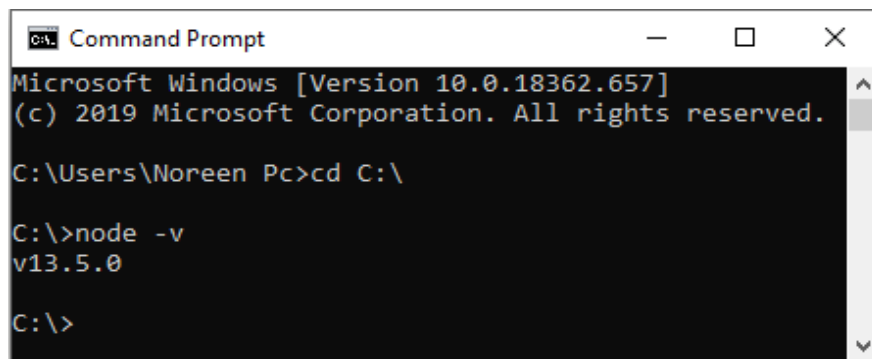
In this section we explain about the languages which we used during implementation. We used Solidity programming language for writing smart contracts on Ethereum blockchain and Node.js is required for running truffle during implementation.

### 2.12.1. Solidity

Solidity is a high level programming language with a Javascript similar script used to write smart contracts. Different developing blockchain platform of solidity are Ethereum, ErisDB, Zeppelin and Counterparty [57]. Contracts written in solidity language are structured similar to the classes in object oriented programming languages [58]. It contains variables, functions, function modifiers, events, structures, and enums which modify these, like in primitive programming [58]. Special variables (msg, block, tx) already defines in solidity that always exist in the global namespace and contain properties to access information about an invocation-transaction and the blockchain [58]. Smart Contracts supports polymorphism and inheritance.

### 2.12.2. Node.js

Node.js can configure your environment for developing smart contracts. NPM (Node Package Manager) which comes with Node.js. For checking if you have already installed node by going to and run following command as shown in Figure 14.



```
Microsoft Windows [Version 10.0.18362.657]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Noreen Pc>cd C:\

C:\>node -v
v13.5.0

C:\>
```

Figure 14: Node.js Version Command

## 2.13. Tools

In this section we explain about the tools which are used in our implementation. Ganache and Truffle are being used in our work for the implementation, compilation and deployment of smart contracts.

### 2.13.1. Ganache

For the development of local blockchain we can use Ganache tool. It provides your own personal blockchain for the development of Ethereum. It will allow you to develop application, deploy smart contracts and also testing of application or smart contract. It's a command line tool and is available for windows, mac and Linux as a desktop application. When you create a workspace on ganache initially there will be ten accounts with different addresses, balance in ethers and mnemonics. Mnemonics contain 12 to 24 words in it, we can call it seed phrase. They allow you to access the cryptocurrency stored in your wallet. It's a secret group of words which cannot be shared with others. We can say Mnemonics is just a memory. The initial view of a workspace is shown in Figure 15.

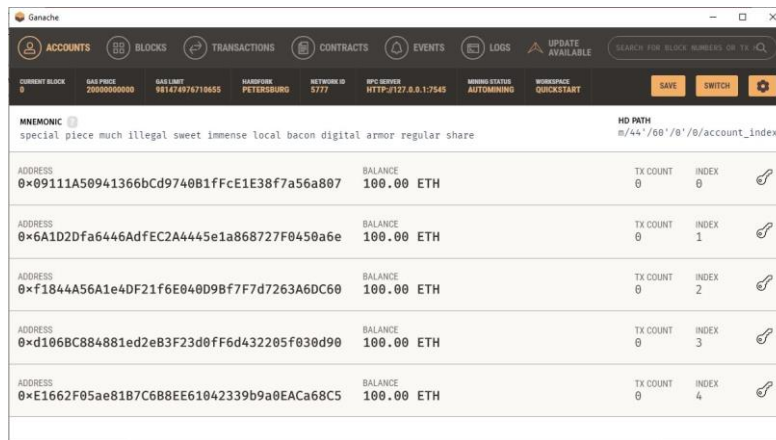


Figure 15: Initial View of Ganache Workspace

### 2.13.2. Truffle

It provides a suite of tools for developing Ethereum smart contracts with the Solidity programming language. You can install Truffle with NPM by using your command line with following command as shown in Figure 16.

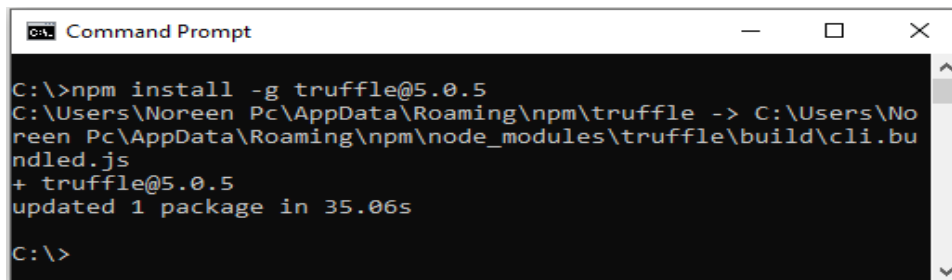


Figure 16: Truffle Version Command

## Chapter 3

# Proposed Methodology

In this chapter, we discuss about our research methodology and our proposed model. The implementation of our model on two different applications are also discuss.

### 3.1. Research Methodology

Our research covers of three steps; literature review, model proposal and then validation as shown in figure 17.

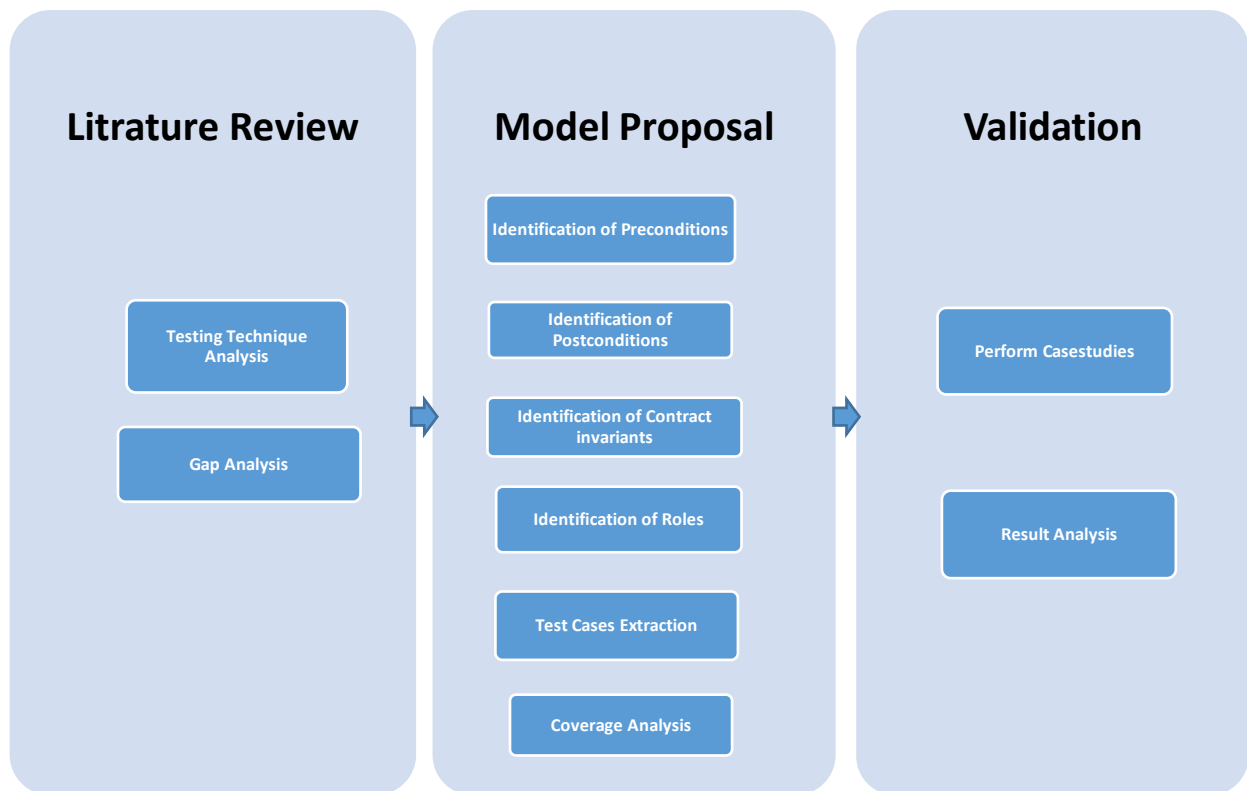


Figure 17: Research Methodology

#### 3.1.1. Literature Review

In this phase we analyse different testing techniques used for the testing of smart contracts. Fully automatic fuzzy engines are already proposed for testing the smart contracts. For testing the vulnerabilities of smart contracts different tools are used and on these tools comparison studies done by few researchers. W.K Chan and Bo Jiang proposed a fuzz testing service named fuse for the testing of Ethereum based smart contracts [16]. Nehai et al. proposed a method to check model

of the application based on smart contracts [78]. The Kernel layer, application layer and the environment layer are the three-fold modelling process on the basis of which a model is build [78]. Translation rules from Solidity to NuSMV language have been provided to build the application layer [78]. Zixin et al. proposed a new tool MuSC for mutation testing of smart contracts [15]. MuSc tool is for Ethereum based smart contracts, it contains all the operators used in Ethereum based smart contact programming language [15]. Fu et al. proposed an automated framework named EVMFuzz to find the security vulnerabilities in Ethereum Virtual Machines (EVMs) developed in different programming languages [19].

### **3.1.2. Model Proposal**

Limited techniques are used to test smart contracts and none of these techniques focuses on pre-conditions and post-conditions which are an essential part of the smart contracts as Nehai et al. [78] proposed way of applying model-checking to a Blockchain Ethereum application based on smart contracts. Few techniques that are used to test runtime behavior of smart contract with application and respond accordingly as Gordan and Joshua [79] proposed a tool CONTRACTLARVA to test smart contracts at run time according to specifications.

As a result of gap analysis six steps model is proposed for testing smart contracts i.e. identification of pre-conditions, identification of post-conditions, identification of contract invariants, identification of roles, test cases extraction and coverage analysis. Based on smart contracts and account information we list all pre-conditions, post-conditions, contract invariants and role properties. In test cases extracting, by considering all mentioned list we write test cases for smart contracts. At the end we analyse defines coverage criteria to confirm the maximum testing coverage.

### **3.1.3. Validation**

On this stage of our research methodology, model is implemented on two applications to show the validity and applicability of the model. The results of applications is analysed on the bases of bugs detected by our model.

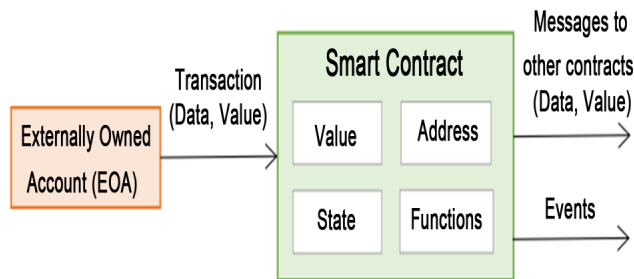
## **3.2. Our Approach**

We study existing testing frameworks for smart contracts and already discussed in this section. Based on literature review we identify the following points on which our model comprises of:

smart contracts structure, layer of blockchain on which smart contract works, Ethereum blockchain, testing technique, test cases generation and the coverage criteria.

### 3.2.1. Smart Contracts structure

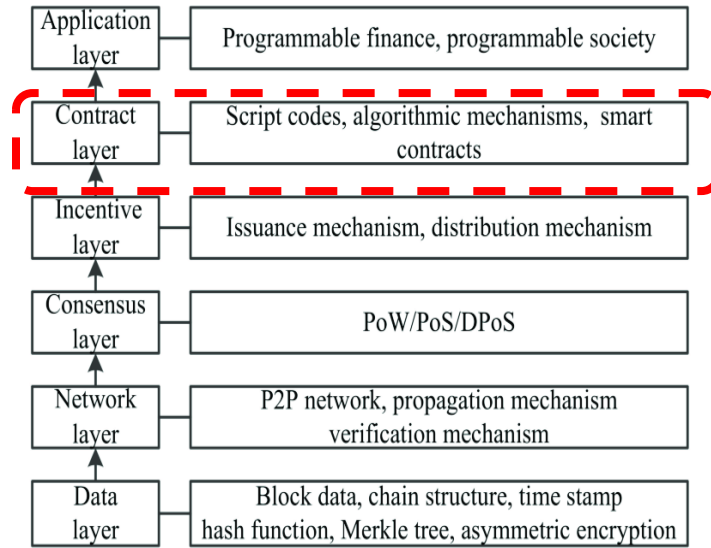
Smart contracts concept was first proposed in 1994 by Nick Szabo [3]. Smart contract is a small piece of code with unique address resides on blockchain [4]. A smart contract contains variables and set of executable functions. Whenever a transaction is executed it contains parameters required for execution of function. On execution of a function, variables state will be changed in smart contract on the basis of logic implemented in function. The structure of smart contract is showed in Figure 18.



**Figure 18:** A basic structure of Smart Contract [3]

### 3.2.2. Layered Architecture of Blockchain

The blockchain architecture is basically divided into six layers (Figure 19): the data layer, the network layer, the consensus layer, the contract layer, the service layer, and the application layer. The data layer and network layer are the lower levels. These layers generate, validate, and store the data and information. The consensus and contract layers are the intermediary between the lower and upper levels. The consensus layer is mainly consists of PoW, PoS, DPoS, and PBFT. The contract layer includes smart contract, consensus protocol, and incentive mechanism. The upper level is at the top of the architecture, including the service and the application platform [38].



**Figure 19:** Blockchain Architecture [38]

### 3.2.3. Ethereum Blockchain

Ethereum is a decentralized public blockchain having a virtual machine named Ethereum Virtual Machine (EVM). EVM is running on all nodes of Ethereum. Ethereum is used to build decentralized application that are not controlled by any central authority. Ethereum practices a blockchain for saving the state of user’s accounts, program codes and its associated states. A node can validate the transaction of another node. Ethereum virtual machine is a computing machine which can be used for the execution of EVM code. EVM code is a byte code. EVM can be used for the validation of transaction details like signature in the transaction, correct number of values and matching of nonce with the specific transaction account nonce. It can also be used for checking the gas is enough or not to execute the transaction. EVM can also transfer ethers from one account to the particular account. EVM also calculate the gas and transaction fee of a transaction to initiate miner’s gas payment.

Smart contracts run on top of the blockchain network and more like the other classes concept in object oriented programming. Smart contracts are exist on Ethereum network independently. Smart contracts byte codes are stored on blockchain. The bytecode contains all the rules and agreements between two parties without the involvement of any third party. As we know smart contracts are consist of different functions then we can say that these functions are must be in the byte code of contract. To keep the database clean and save the space Ethereum allows smart contracts to automatically self-destroy when they are not doing any operations or saving values.

### 3.2.4. Testing Technique

We propose a model-based testing technique named model-based smart contracts testing technique. We discover four model aspects:

- Pre-conditions
- Post-conditions
- Contract invariant
- Contract roles

Once we have considered model-based aspects given above, we turn to white-box testing concepts since we have access to code. We define how do we extract them and how do we develop test cases pertaining to these model aspects.

- **Pre-Condition**

The condition which is required before to execute the function of a smart contract. It can be last executed step, last executed function output and accessibility of existing data. The pre-conditions required for the execution of smart contracts are account with sufficient balance, address of the smart contract and inputs for calling function of smart contract. If any condition is false then smart contract will not be executed and state will not update as shown in Table 2. We list all pre-conditions that must be met before a contract can be executed. This is done using Table 1.

**Table 1:** Pre-conditions

<b>S No.</b>	<b>Pre-condition</b>	<b>Justification</b>	<b>Inputs (Set)</b>
<b>1.</b>	Account with sufficient balance	Account required for transaction and balance is use for paying gas fee on processing.	Account address, balance
<b>2.</b>	Address	Deployed address of smart contract required for interaction	Account address
<b>3.</b>	Inputs	Functions with some input values are required for triggering smart contracts.	Account address, balance

In order to develop test cases from identified pre-conditions, we additionally consider account related information and we characterize them into test cases with positive and negative intent considering input spaces. This is shown in Table-2.

**Table 2:** Test cases related to Pre-conditions

S No.	Account with Sufficient Balance	Address	Inputs	State (Expected Output)
1.	✓	✓	✓	Updated
2.	✗	✓	✓	Not Updated
3.	✓	✗	✓	Not Updated
4.	✓	✓	✗	Not Updated

- **Post-condition**

Post-condition is the desired output, desired effect on data after the complete execution of a function, relative to the given parameters that satisfies the pre-conditions of a function. The Post-conditions for a smart contract are: after the execution transaction should be successful, current state of the smart contract which is going to be changed and on transaction there must be an event triggered. If any condition is false then the state of smart contract will not be update as shown in Table 4. We list all post-conditions that must be met before a contract can be executed. This is done using Table 3.

**Table 3:** Post-conditions

S No.	Pre-condition	Justification	Inputs (Set)
1.	Transaction	After execution of smart contract transaction will be done	Balance update
2.	Current state	State of the smart contract is required for updating	Storage, address
3.	Events	Event triggered on a transaction execution	Balance transfer event



In order to develop test cases from identified post-conditions, we additionally consider smart contract related information and we characterize them into test cases with positive and negative intent considering input spaces. This is shown in Table-4.

**Table 4:** Test cases related to Post-conditions

S No.	Transaction	Current State	Events	Next State (Post-condition for previous)
1.	✓	✓	✓	Update
2.	✗	✓	✓	Not Update
3.	✓	✗	✓	Not Update
4.	✓	✓	✗	Not Update

- **Contract Invariants**

The condition which is required to be true before and after the iteration of a loop. The state of smart contract, its address and inputs are the contract invariants required for the execution of a smart contract. If any condition is false then the smart contract will not execute as shown in Table 6. We list all contract invariants classes that must be met before a contract can be executed. This is done using Table 5.

**Table 5:** Contract Invariants Classes

S No.	Pre-condition	Justification	Inputs (Set)
1.	State	Smart contract must have a state	Storage, address
2.	Address	After deployment smart contract have an address for external users to communicate	Smart contract address
3.	Inputs	For smart contract execution there must be some input values	Account address, balance

In order to develop test cases from identified contracts invariants classes, we additionally consider smart contract related information and we characterize them into test cases with positive and negative intent considering input spaces. This is shown in Table-6.

**Table 6:** Test cases related to Contract Invariants Classes

S No	State	Address	Inputs	Execute
1.	✓	✓	✓	✓
2.	✗	✓	✓	✗
3.	✓	✗	✓	✗
4.	✓	✓	✗	✗

- **Testing Each Role**

All the actors/users roles interact with smart contract through application should be tested. Account with sufficient balance and inputs are required for testing the role. If any condition is false then smart contract will not execute as shown in Table 8. We list all roles properties that must be met before a contract can be executed. This is done using Table 7.

**Table 7:** Roles

S No.	Pre-condition	Justification	Inputs (Set)
1.	Accounts with sufficient balance	Account required for transaction and balance is use for paying gas fee on processing.	Account address, balance
2.	Inputs	For interaction with smart contract there must be some input values	Account address, balance
3.	Role	must be a registered or authentic role for smart contracts interaction	Owner

In order to develop test cases from identified role properties, we additionally consider roles related information and we characterize them into test cases with positive and negative intent considering input spaces. This is shown in Table-8.

**Table 8:** Test cases related to Roles

S No.	Account with Sufficient Balance	Inputs	Role	Expected Output
1.	✓	✓	✓	✓
2.	✗	✓	✗	✗
3.	✓	✗	✗	✗

### 3.2.5. Coverage Criteria

We propose a coverage criterion in term of our model translating them into test requirements. Therefore, we say that a test suite is providing coverage with respect to test requirements which we define in terms of all pre-conditions coverage, all post-conditions coverage, all contract invariants coverage and all roles coverage.

- **Pre Conditions**

We ensure all pre-conditions are executed with all options as shown in Table 2 above:

- **Post-conditions**

We ensure all post-conditions are executed with all options as shown in Table 4 above:

- **Contract Invariants**

We ensure all contract invariants are executed with all options as shown in Table 6 above:

- **Testing Each Role**

We ensure all roles are executed with all options as shown in Table 8 above:

### 3.3. Running Applications

We take two open source decentralized applications from internet with frontend and backend.

#### 3.3.1. Case Study 1:

We take an application code from internet which is a DApp with frontend and backend. This application is based on supply chain management for cheese. This application tracks cheese sale from farm to the customer. All actors have to use this application according to their roles. The business logic of this application is written in smart contracts which will save all the data. As we discussed in previous section that Ethereum facilitate developers to write any type of contract, it

is not necessary that our business logic is based on money. In this application, smart contracts are written to store data of each step of sales or purchase process. Here, we have the following smart contracts:

1. ConsumerRole
2. DistributorRole
3. FarmerRole
4. RetailerRole
5. Ownable
6. SupplyChain

The following are the pre-conditions for smart-contracts on the bases of different roles:

<b>S No.</b>	<b>Roles</b>	<b>Pre-conditions</b>
<b>1.</b>	Farmer	<ul style="list-style-type: none"> <li>• Authentic account</li> <li>• Balance</li> <li>• UPC</li> <li>• Farm name</li> <li>• Farm address</li> <li>• Farm latitude</li> <li>• Farm longitude</li> <li>• Product notes</li> <li>• Product price</li> </ul>
<b>2.</b>	Distributor	<ul style="list-style-type: none"> <li>• Authentic account</li> <li>• Balance</li> <li>• UPC</li> <li>• Product price</li> <li>• Total slices</li> </ul>
<b>3.</b>	Retailer	<ul style="list-style-type: none"> <li>• Authentic account</li> <li>• Balance</li> <li>• UPC</li> <li>• Product price</li> </ul>
<b>4.</b>	Consumer/Customer	<ul style="list-style-type: none"> <li>• Authentic account</li> <li>• Balance</li> <li>• UPC</li> </ul>

The following are the post-conditions for smart-contracts:

S No.	Roles	Post-conditions
1.	Farmer	<ul style="list-style-type: none"> <li>Farmer sell cheese</li> <li>Distributor buy cheese</li> <li>Farmer ship cheese to the distributor</li> <li>Balance amount increases</li> </ul>
2.	Distributor	<ul style="list-style-type: none"> <li>Able to process cheese</li> <li>Distributor sell cheese</li> <li>Retailer buy cheese</li> <li>Distributor ship cheese</li> <li>Balance amount increases</li> </ul>
3.	Retailer	<ul style="list-style-type: none"> <li>Retailer process cheese</li> <li>Retailer sell cheese</li> <li>Consumer/Customer buy cheese</li> <li>Balance amount increases</li> </ul>
4.	Consumer/Customer	<ul style="list-style-type: none"> <li>Consumer purchase cheese</li> <li>Balance amount deducted as per product price</li> </ul>

The following are the contract invariants classes required for smart contracts testing.

S No.	Contract Invariants
1.	State
2.	Address of the deployed smart contract
3.	Inputs

Actors (we identify them as Roles) of this application are as following:

- Farmer
- Distributor
- Retailer
- Consumer/Customer

### 3.3.1.1. Smart Contracts

There are total six smart contracts and a library written for this application. All role based smart contracts import the library named with Roles. All other smart contracts other than library are as following:

- **ConsumerRole**

All the functions related to consumer are written in this smart contract to handle the consumer or customer role. Functions related to consumer role are adding, removing and checking. Add function is written to add new consumer, remove function is written for deleting a consumer while checking function is written to check that the consumer is an authentic user or not. The code of this contract is shown in Figure 20.

```
pragma solidity >=0.4.24;
// Import the library 'Roles'
import "./Roles.sol";

contract ConsumerRole {
    using Roles for Roles.Role;
    // Define 2 events, one for Adding, and other for Removing
    event ConsumerAdded(address indexed account);
    event ConsumerRemoved(address indexed account);

    // Define a struct 'consumers' by inheriting from 'Roles' library, struct Role
    Roles.Role private consumers;
    // In the constructor make the address that deploys this contract the 1st consumer
    constructor() public {
        _addConsumer(msg.sender);
    }
}
```

**Figure 20:** Consumer Role Code

- **DistributorRole**

All the functions related to distributor are written in this smart contract to handle the distributor role. Functions related to distributor role are adding, removing and checking. Add function is written to add new distributor, remove function is written for deleting a distributor while checking function is written to check that the distributor is an authentic user or not. The code of this contract is shown in Figure 21.

```
pragma solidity >=0.4.24;
import "./Roles.sol";

// Define a contract 'DistributorRole' to manage this role - add, remove, check
contract DistributorRole {
    using Roles for Roles.Role;
    // Define 2 events, one for Adding, and other for Removing
    event DistributorAdded(address indexed account);
    event DistributorRemoved(address indexed account);
    // Define a struct 'distributors' by inheriting from 'Roles' library, struct Role
    Roles.Role private distributors;
    // In the constructor make the address that deploys this contract the 1st distributor
    constructor() public {
        _addDistributor(msg.sender);
    }
}
```

**Figure 21:** Distributor Role Code

- **FarmerRole**

All the functions related to farmer are written in this smart contract to handle the farmer role. Functions related to farmer role are adding, removing and checking. Add function is written to add new farmer, remove function is written for deleting a farmer while checking function is written to check that the farmer is an authentic user or not. The code of this contract is shown in Figure 22.

```
pragma solidity >=0.4.24;

import "./Roles.sol";
contract FarmerRole {
    using Roles for Roles.Role;

    event FarmerAdded(address indexed account);
    event FarmerRemoved(address indexed account);
    Roles.Role private farmers;

    // In the constructor make the address that deploys this contract the 1st farmer
    constructor() public {
        _addFarmer(msg.sender);
    }
}
```

**Figure 22:** Farmer Role Code

- **RetailerRole**

All the functions related to retailer are written in this smart contract to handle the retailer role. Functions related to retailer role are adding, removing and checking. Add function is written to add new retailer, remove function is written for deleting a retailer while checking function is written to check that the retailer is an authentic user or not. The code of this contract is shown in Figure 23.

```
pragma solidity >=0.4.24;
import "./Roles.sol";
contract RetailerRole {
    using Roles for Roles.Role;

    // Define 2 events, one for Adding, and other for Removing
    event RetailerAdded(address indexed account);
    event RetailerRemoved(address indexed account);

    // Define a struct 'retailers' by inheriting from 'Roles' library, struct Role
    Roles.Role private retailers;
    // In the constructor make the address that deploys this contract the 1st retailer
    constructor() public {
        _addRetailer(msg.sender);
    }
}
```

**Figure 23:** Retailer Role Code

- **Ownable**

This contract is written to authenticate the user and assigning roles to the user according to their interaction with the main contract (shown in Figure 24).

```
pragma solidity >=0.4.24;

/// Provides basic authorization control
contract Ownable {
    address private origOwner;
    // Define an Event
    event TransferOwnership(address indexed oldOwner, address indexed newOwner);

    /// Assign the contract to an owner
    constructor () internal {
        origOwner = msg.sender;
        emit TransferOwnership(address(0), origOwner);
    }
}
```

**Figure 24:** Ownable Code

- **SupplyChain**

It's a main contract which inherit all other smart contracts. All main functions and events are written in this contract. This contract address is used for the external system communication with the smart contract. The code of this contract is shown in Figure 25.

```
pragma solidity >=0.4.24;

// inherited contracts
import './pamigianocore/Ownable.sol';
import './pamigianoaccesscontrol/FarmerRole.sol';
import './pamigianoaccesscontrol/DistributorRole.sol';
import './pamigianoaccesscontrol/RetailerRole.sol';
import './pamigianoaccesscontrol/ConsumerRole.sol';

contract SupplyChain is Ownable,FarmerRole,DistributorRole,RetailerRole,ConsumerRole {

    ...

    function produceltemByFarmer(uint _upc, string memory _originFarmName, string memory
    _originFarmInformation, string memory _originFarmLatitude, string memory _originFarmLongitude,
    string memory _productNotes, uint _price) public
    onlyFarmer() // check address belongs to farmerRole
    { ... }
}
```

**Figure 25:** Supply Chain Code



### **3.3.1.2. Interaction between Roles**

We design a sequence diagram on the bases of our actor's communication with smart contracts as shown in Figure 26. We clearly mentioned all the called function of smart contracts in diagram. As we mentioned in previous detail that if any external system have to communicate with smart contract than address of SupplyChain contract will be used. SupplyChain contract is a main contract which inherit other smart contracts. We can say that other smart contracts are like model classes. All smart contracts are deployed on Ethereum network but for communication SupplyChain smart contract is used. The main purpose of developing this application is to manage the supply chain of cheese and automate every process. We have four actors for our application which are farmer, distributor, retailer and consumer/customer.

Farmer produce cheese in his farm and sell it to distributor. When he wants to sell his cheese, three functions of smart contracts is called to trigger smart contract which are: `produceItemByFarmer()`, `sellItemByFarmer()` and `shippedItemByFarmer()`. Input variables for all these functions are as following:

- `produceItemByFarmer()`: Universal product code (UPC), `originFarmName`, `originFarmInformation`, `originFarmLatitude`, `originFarmLongitude`, `productNotes` and price
- `sellItemByFarmer()`: Universal product code (UPC) and price
- `shippedItemByFarmer()`: Universal product code (UPC)

Distributor buy cheese from farmer and sell it to retailers. When he wants to sell his cheese, six functions of smart contracts is called to trigger smart contract which are: `purchaseItemByDistributor()`, `receivedItemByDistributor()`, `processedItemByDistributor()`, `packageItemByDistributor()`, `sellItemByDistributor()` and `shippedItemByDistributor()`. Input variables for all these functions are as following:

- `purchaseItemByDistributor()`: Universal product code (UPC)
- `receivedItemByDistributor()`: Universal product code (UPC)
- `processedItemByDistributor()`: Universal product code (UPC) and slices
- `packageItemByDistributor()`: Universal product code (UPC)
- `sellItemByDistributor()`: Universal product code (UPC) and price
- `shippedItemByDistributor()`: Universal product code (UPC)

Retailer buy cheese from distributor and sell it to consumers/customers. When he wants to sell his cheese, three functions of smart contracts is called to trigger smart contract which are: `purchaseItemByRetailer()`, `receivedItemByRetailer()` and `sellItemByRetailer()`. Input variables for all these functions are as following:

- `purchaseItemByRetailer()`: Universal product code (UPC)
- `receivedItemByRetailer()`: Universal product code (UPC)
- `sellItemByRetailer()`: Universal product code (UPC) and price

Consumer/customer buy cheese from retailer. On buying cheese by customer from retailer a functions of smart contracts is called to trigger smart contract is `purchaseItemByConsumer()`. Input variables for this functions are as following:

- `purchaseItemByConsumer()`: Universal product code (UPC)

`FetchItemBufferOne()`, `fetchItemBufferTwo()` and `fetchitemHistory()` are three general functions which can be called by any role. We shows all these functions calling with only consumer/customer role but it can be called by anyone to see the data of product.

- `fetchItemBufferOne()`: Universal product code (UPC)
- `fetchItemBufferTwo()`: Universal product code (UPC)
- `fetchitemHistory()`: Universal product code (UPC)

On the basis of defined coverage criteria and considering sequence diagram, we write test cases file for smart contract in JavaScript. We write a truffle file to run the test file. At the end we design a testing matrix with Pre-condition, Post-condition, expected result, output and result. We use Ganache, Truffle and Git for our testing.

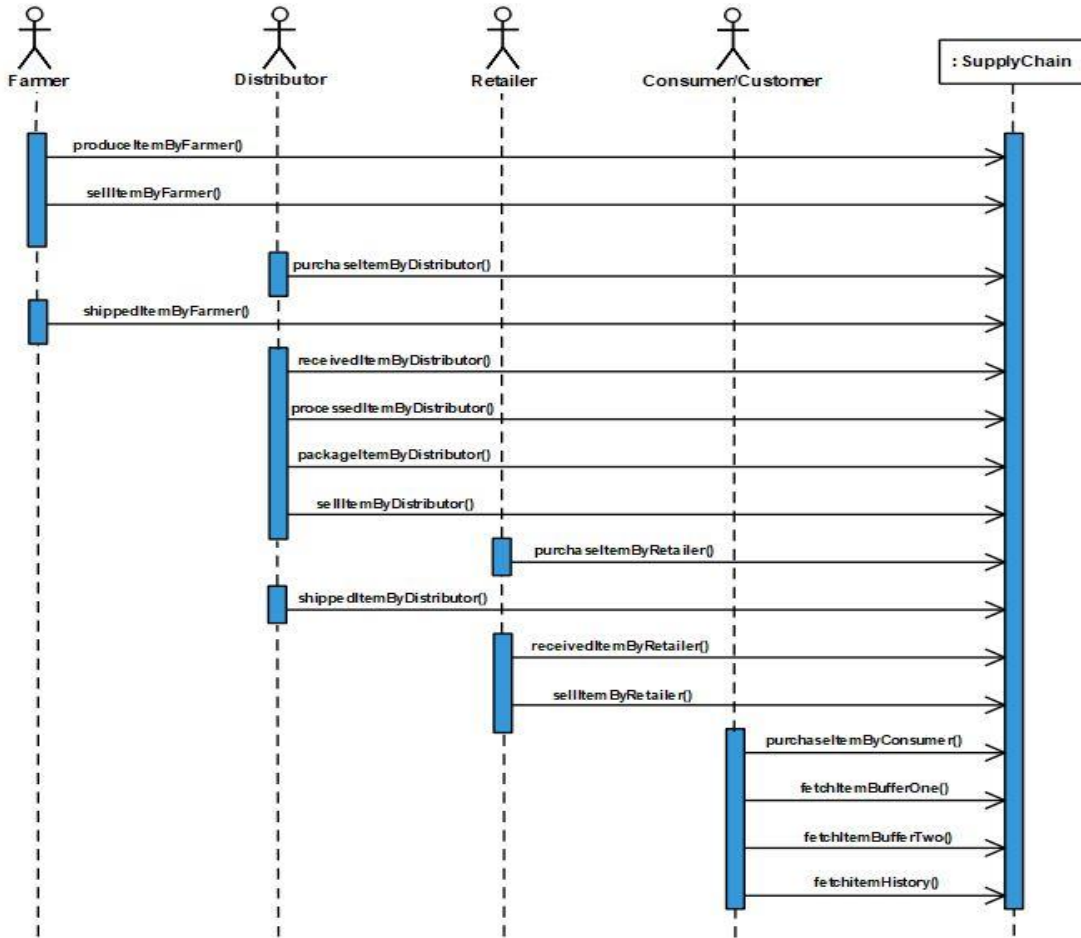


Figure 26: Sequence Diagram

### 3.3.1.3. Test Cases Extraction

We divide our traceability matrix containing test cases according to part identified in our proposal. In our application every previous function is the Pre-condition of next function and the output is the Post-condition of that function. In case of roles, they must have an authentic account to process with sufficient balance. The contract invariants are successfully tested when any function of the smart contract is executed with some input values, its state is changed on successful transaction and after deployment we have an address of the smart contract by which we are doing interaction. In our traceability matrix, post-condition and pre-condition are for individual test case.

S No	Test Case	Pre-Condition	Post-Condition	Output	Expected Results	Actual Results(Pass/Fail)
1.	Farmer produce cheese	Must have an account with all detail (UPC,	Farmer will be allowed to produce cheese	Farmer allowed to	Farmer allowed to produce cheese	Pass

		originFarmName, originFarmInformation, originFarmLatitude, originFarmLongitude, productNotes, productPrice)		produce cheese		
2.	farmer produce cheese with null UPC	Must have an account with all detail (UPC, originFarmName, originFarmInformation, originFarmLatitude, originFarmLongitude, productNotes, productPrice)	Farmer will not be allowed to produce cheese	Farmer not allowed to produce cheese	Farmer not allowed to produce cheese	<b>Pass</b>
3.	farmer produce cheese with null Farmer name	Must have an account with all detail (UPC, originFarmName, originFarmInformation, originFarmLatitude, originFarmLongitude, productNotes, productPrice)	Farmer will not be allowed to produce cheese	Farmer not allowed to produce cheese	Farmer not allowed to produce cheese	<b>Pass</b>
4.	farmer produce cheese with null Farmer Address	Must have an account with all detail (UPC, originFarmName, originFarmInformation,	Farmer will not be allowed to produce cheese	Farmer not allowed to produce cheese	Farmer not allowed to produce cheese	<b>Pass</b>

		originFarmLatitude, originFarmLongitude, productNotes, productPrice)				
5.	farmer produce cheese with null Latitude	Must have an account with all detail (UPC, originFarmName, originFarmInformation, originFarmLatitude, originFarmLongitude, productNotes, productPrice)	Farmer will not be allowed to produce cheese	Farmer not allowed to produce cheese	Farmer not allowed to produce cheese	<b>Pass</b>
6.	farmer produce cheese with null longitude	Must have an account with all detail (UPC, originFarmName, originFarmInformation, originFarmLatitude, originFarmLongitude, productNotes, productPrice)	Farmer will not be allowed to produce cheese	Farmer not allowed to produce cheese	Farmer not allowed to produce cheese	<b>Pass</b>
7.	farmer produce cheese with product notes	Must have an account with all detail (UPC, originFarmName, originFarmInformation, originFarmLatitude, originFarmLongitude,	Farmer will not be allowed to produce cheese	Farmer not allowed to produce cheese	Farmer not allowed to produce cheese	<b>Pass</b>

		productNotes, productPrice)				
<b>8.</b>	farmer produce cheese with null product price	Must have an account with all detail (UPC, originFarmName, originFarmInformation, originFarmLatitude, originFarmLongitude, productNotes, productPrice)	Farmer will not be allowed to produce cheese	Farmer not allowed to produce cheese	Farmer not allowed to produce cheese	<b>Pass</b>
<b>9.</b>	farmer produce cheese with null address	Must have an account with all detail (UPC, originFarmName, originFarmInformation, originFarmLatitude, originFarmLongitude, productNotes, productPrice)	Farmer will not be allowed to produce cheese	Farmer not allowed to produce cheese	Farmer not allowed to produce cheese	<b>Pass</b>
<b>10.</b>	farmer sell cheese	Must have an account with all detail (UPC and product price )	Farmer will be allow to sell cheese.	Allows farmer to sell cheese.	Allows farmer to sell cheese	<b>Pass</b>
<b>11.</b>	Farmer sell cheese with null UPC.	Must have an account with all detail (UPC and product price )	Farmer will not be allow to sell cheese.	Farmer not allow selling cheese.	Farmer not allow selling cheese.	<b>Pass</b>
<b>12.</b>	farmer sell cheese with null product price	Must have an account with all detail (UPC and product price )	Farmer will not be allow to sell cheese.	Farmer not allow selling cheese.	Farmer not allow selling cheese.	<b>Pass</b>

13.	farmer sell cheese with null address	Must have an account with all detail (UPC and product price )	Farmer will not be allow to sell cheese.	Farmer not allow selling cheese.	Farmer not allow selling cheese.	<b>Pass</b>
14.	distributor buy cheese	Must have an account with UPC and balance	Distributor will buy cheese	Distributor buys cheese	Distributor buys cheese	<b>Pass</b>
15.	Distributor buy cheese with null UPC	Must have an account with UPC and balance	Distributor will not buy cheese	Distributor not allowed to buys cheese	Distributor not allowed to buys cheese	<b>Pass</b>
16.	Distributor buy cheese with 0 balance	Must have an account with UPC and balance	Distributor will not buy cheese	Distributor not allowed to buys cheese	Distributor not allowed to buys cheese	<b>Pass</b>
17.	Distributor buy cheese with null address	Must have an account with UPC and balance	Distributor will not buy cheese	Distributor not allowed to buys cheese	Distributor not allowed to buys cheese	<b>Pass</b>
18.	Farmer ship cheese	Must have an account with UPC	Allows farmer to ship cheese	Farmer allowed to ship cheese	Farmer allowed to ship cheese	<b>Pass</b>
19.	Farmer ship cheese with null UPC	Must have an account with UPC	Farmer will not be allowed to ship cheese	Farmer not allowed to ship cheese	Farmer not allowed to ship cheese	<b>Pass</b>
20.	Farmer ship cheese with null address	Must have an account with UPC	Farmer will not be allowed to ship cheese	Farmer not allowed to ship cheese	Farmer not allowed to ship cheese	<b>Pass</b>
21.	Distributor receive cheese	Must have an account with UPC	Distributor will receive cheese	Distributor receives cheese	Distributor receives cheese	<b>Pass</b>
22.	Distributor receive cheese	Must have an account with UPC	Distributor will not receive cheese	Distributor not receive	Distributor not	<b>Pass</b>

	with null UPC			d cheese	received cheese	
23.	Distributor or receive cheese with null address	Must have an account with UPC	Distributor will not receive cheese	Distributor not received cheese	Distributor not received cheese	<b>Pass</b>
24.	Allows distributor to process cheese	Must have an account with UPC and total slices	Distributor will be allowed to process cheese	Distributor allowed to process cheese	Distributor allowed to process cheese	<b>Pass</b>
25.	Allows distributor to process cheese with null UPC	Must have an account with UPC and total slices	Distributor will not be allowed to process cheese	Distributor not allowed to process cheese	Distributor not allowed to process cheese	<b>Pass</b>
26.	Allows distributor to process cheese with 0 slices	Must have an account with UPC and total slices	Distributor will not be allowed to process cheese	Distributor not allowed to process cheese	Distributor not allowed to process cheese	<b>Pass</b>
27.	Allows distributor to process cheese with null slices	Must have an account with UPC and total slices	Distributor will not be allowed to process cheese	Distributor not allowed to process cheese	Distributor not allowed to process cheese	<b>Pass</b>
28.	Allows distributor to process cheese with null address	Must have an account with UPC and total slices	Distributor will not be allowed to process cheese	Distributor not allowed to process cheese	Distributor not allowed to process cheese	<b>Pass</b>
29.	Distributor or pack cheese	Must have an account with UPC	Allows distributor for cheese packaging	Distributor allowed for cheese	Distributor allowed for cheese packaging	<b>Pass</b>



				packaging		
<b>30.</b>	Distributor or pack cheese with null UPC	Must have an account with UPC	distributor will not be allowed for cheese packaging	Distributor not allowed for cheese packaging	Distributor not allowed for cheese packaging	<b>Pass</b>
<b>31.</b>	Distributor or pack cheese with null address	Must have an account with UPC	distributor will not be allowed for cheese packaging	Distributor not allowed for cheese packaging	Distributor not allowed for cheese packaging	
<b>32.</b>	Distributor or sell cheese	Must have an account with UPC and product price	Distributor will be able to sell cheese	Distributor sell cheese	Distributor sell cheese	<b>Pass</b>
<b>33.</b>	Distributor or sell cheese with null UPC	Must have an account with UPC and product price	Distributor will not be able to sell cheese	Distributor not sell cheese	Distributor not sell cheese	<b>Pass</b>
<b>34.</b>	Distributor or sell cheese with 0 product price	Must have an account with UPC and product price	Distributor will not be able to sell cheese	Distributor not sell cheese	Distributor not sell cheese	<b>Pass</b>
<b>35.</b>	Distributor or sell cheese with null product price	Must have an account with UPC and product price	Distributor will not be able to sell cheese	Distributor not sell cheese	Distributor not sell cheese	<b>Pass</b>
<b>36.</b>	Distributor or sell cheese with null address	Must have an account with UPC and product price	Distributor will not be able to sell cheese	Distributor not sell cheese	Distributor not sell cheese	<b>Pass</b>

37.	Retailer purchase Cheese	Must have an account with UPC and balance	Retailer will purchase cheese	Retailer purchases cheese	Retailer purchases cheese	<b>Pass</b>
38.	Retailer purchase Cheese with null UPC	Must have an account with UPC and balance	Retailer will not purchase cheese	Retailer not allowed for purchasing cheese	Retailer not allowed for purchasing cheese	<b>Pass</b>
39.	Retailer purchase Cheese with 0 balance	Must have an account with UPC and balance	Retailer will not purchase cheese	Retailer not allowed for purchasing cheese	Retailer not allowed for purchasing cheese	<b>Pass</b>
40.	Retailer purchase Cheese with null account	Must have an account with UPC and balance	Retailer will not purchase cheese	Retailer not allowed for purchasing cheese	Retailer not allowed for purchasing cheese	<b>Pass</b>
41.	Distributor or ship cheese	Must have an account with UPC	Distributor will be allowed to ship cheese	Distributor allowed to ship cheese	Distributor allowed to ship cheese	<b>Pass</b>
42.	Distributor or ship cheese with null UPC	Must have an account with UPC	Distributor will not be allowed to ship cheese	Distributor not allowed to ship cheese	Distributor not allowed to ship cheese	<b>Pass</b>
43.	Distributor or ship cheese with null address	Must have an account with UPC	Distributor will not be allowed to ship cheese	Distributor not allowed to ship cheese	Distributor not allowed to ship cheese	<b>Pass</b>
44.	Allows retailer to receive cheese	Must have an account with UPC	Retailer will be allowed to receive cheese	Retailer allowed to receive cheese	Retailer allowed to receive cheese	<b>Pass</b>

45.	Allows retailer to receive cheese with null UPC	Must have an account with UPC	Retailer will not be allowed to receive cheese	Retailer not allowed to receive cheese	Retailer not allowed to receive cheese	<b>Pass</b>
46.	Allows retailer to receive cheese with null address	Must have an account with UPC	Retailer will not be allowed to receive cheese	Retailer not allowed to receive cheese	Retailer not allowed to receive cheese	<b>Pass</b>
47.	Allows a retailer to sell cheese	Must have an account with UPC and product price	Retailer will be allowed to sell cheese	Retailer allowed to sell cheese	Retailer allowed to sell cheese	<b>Pass</b>
48.	Allows a retailer to sell cheese with null UPC	Must have an account with UPC and product price	Retailer will not be allowed to sell cheese	Retailer not allowed to sell cheese	Retailer not allowed to sell cheese	<b>Pass</b>
49.	Allows a retailer to sell cheese with 0 product price	Must have an account with UPC and product price	Retailer will not be allowed to sell cheese	Retailer not allowed to sell cheese	Retailer not allowed to sell cheese	<b>Pass</b>
50.	Allows a retailer to sell cheese with null product price	Must have an account with UPC and product price	Retailer will not be allowed to sell cheese	Retailer not allowed to sell cheese	Retailer not allowed to sell cheese	<b>Pass</b>
51.	Allows a retailer to sell cheese with null address	Must have an account with UPC and product price	Retailer will not be allowed to sell cheese	Retailer not allowed to sell cheese	Retailer not allowed to sell cheese	<b>Pass</b>
52.	Allows a consumer to	Must have an account with UPC and balance	Consumer will be allowed for	Consumer allowed to	Consumer allowed to purchase cheese	<b>Pass</b>

	purchase cheese		purchasing cheese	purchase cheese		
53.	Allows a consumer to purchase cheese with null UPC	Must have an account with UPC and balance	Consumer will not be allowed for purchasing cheese	Consumer not allowed to purchase cheese	Consumer not allowed to purchase cheese	<b>Pass</b>
54.	Allows a consumer to purchase cheese with 0 balance	Must have an account with UPC and balance	Consumer will not be allowed for purchasing cheese	Consumer not allowed to purchase cheese	Consumer not allowed to purchase cheese	<b>Pass</b>
55.	Allows a consumer to purchase cheese with null account	Must have an account with UPC and balance	Consumer will not be allowed for purchasing cheese	Consumer not allowed to purchase cheese	Consumer not allowed to purchase cheese	<b>Pass</b>
56.	fetchItemBufferOn() test	Must have an UPC	Item will be buffered successfully	Item buffered successfully	Item buffered successfully	<b>Pass</b>
57.	fetchItemBufferOn() test with null UPC	Must have an UPC	Item will not be buffered successfully	Item not buffered successfully	Item not buffered successfully	<b>Pass</b>
58.	fetchItemBufferTwo() test	Must have an UPC	Item will be buffered successfully	Item not buffered successfully	Item not buffered successfully	<b>Pass</b>
59.	fetchItemBufferTwo() test with null UPC	Must have an UPC	Item will be buffered successfully	Item not buffered	Item not buffered successfully	<b>Pass</b>

				success fully		
60.	fetchItem History() test	Must have an UPC	Item history will be shows	Item history showed	Item history showed	Pass
61.	fetchItem History() with null UPC	Must have an UPC	Item history will not be shows	Item history not showed	Item history not showed	Pass
62.	Add distributor account as an farmer account	Must have an account with balance	Distributor account will not be added as farmer account	Distributor account success fully added as farmer account	Distributor account will not be added as farmer account	Fail
63.	UPC less than 12 digits is not acceptable	Must have an account and UPC with all other inputs	UPC with 1 digit will not be acceptable	Transaction done success fully with 1 digit UPC	UPC with 1 digit will not be acceptable	Fail
64.	Add distributor account as an farmer account	Must have an account with balance	Distributor account will not be added as farmer account	Distributor account success fully added as farmer account	Distributor account will not be added as farmer account	Fail
65.	UPC less than 12 digits is not acceptable	Must have an account and UPC with all other inputs	UPC with 1 digit will not be acceptable	Transaction done success fully with 1 digit UPC	UPC with 1 digit will not be acceptable	Fail
66.	Farmer name length	Must have an account with	Farmer name with one digit	Cheese detail added	Farmer name with one digit	Fail

	should be more than two characters	all other inputs	will not be acceptable	successfully with one digit farmer name	will not be acceptable	
67.	Farmer is adding more cheese information	Must have an account with all other inputs	Farmer will be able to add more cheese	Transaction cancelled	Farmer will be able to add more cheese	<b>Fail</b>
68.	Farm name length should be more than one character	Must have an account with all other inputs	Farm name with one digit will not be acceptable	Cheese detail added successfully with one digit farm name	Farm name with one digit will not be acceptable	<b>Fail</b>
69.	Cheese added by Distributor	Must have an account with all other inputs	Distributor account will not be able to add cheese	Distributor account successfully added cheese	Distributor account will not be able to add cheese	<b>Fail</b>
70.	Cheese added by Retailer	Must have an account with all other inputs	Retailer account will not be able to add cheese	Retailer account successfully added cheese	Retailer account will not be able to add cheese	<b>Fail</b>
71.	Cheese added by consumer	Must have an account with all other inputs	Consumer account will not be able to add cheese	Consumer account successfully added cheese	Consumer account will not be able to add cheese	<b>Fail</b>

### 3.3.2. Case study 2

We take an application code from internet which is a DApp with frontend and backend. This application is based on products sale and purchase. This application tracks product sale from sale to the purchase process. All actors have to use this application according to their roles. The business logic of this application is written in smart contracts which will save all the data. Here, we have the following smart contracts:

- AddPurchaseProducts

The following are the pre-conditions for smart-contracts on the bases of different roles:

S No.	Roles	Pre-conditions
1.	Seller	<ul style="list-style-type: none"><li>• Authentic Account</li><li>• Balance</li><li>• Product Name</li><li>• Product Price</li></ul>
2.	Buyer	<ul style="list-style-type: none"><li>• Authentic Account</li><li>• Balance</li><li>• Product id</li></ul>

The following are the post-conditions for smart-contracts on the bases of different roles:

S No.	Roles	Pre-conditions
1.	Seller	<ul style="list-style-type: none"><li>• Product Added</li><li>• Balance increase</li></ul>
2.	Buyer	<ul style="list-style-type: none"><li>• Product purchased</li><li>• Deducted balance</li></ul>

The following are the contract invariants classes required for smart contracts testing.

S No.	Contract Invariants Classes
1.	State
2.	Address of the deployed smart contract
3.	Inputs

Actors (we identify them as Roles) of this application are as following:

- Seller
- Buyer

### **3.3.2.1.    *Test cases Extraction***

We divide our traceability matrix containing test cases according to part identified in our proposal (See Appendix A for traceability matrix).



## Chapter 4

# Results and Evaluation

In this chapter we will discuss our case studies results further we map our test cases of both case studies on our defined model. We do a short comparison of our model technique with different previous work techniques and the validation of our test cases result are also discussed.

### 4.1. Testing Results

After applying our proposed model the testing results of cases study 1 and 2 testcases are discussed in this section.

#### 4.1.1. Case Study 1

We extract 71 test cases for our smart contract from which 61 test cases are pass and 10 test cases fails as shown below.

**Total Test cases = 71**

**Pass Test cases = 61**

**Fail Test cases = 10**

Test cases fails because of some reasons, which are as following:

- Universal Product Code is standard 12 digits number but smart contract is accepting one digit UPC too.
- Doing transactions successfully with 1 character farmer name
- Farmer not able to add cheese details again after one attempt.
- Roles are not properly define.

#### 4.1.2. Case Study 2

We extract 13 test cases for our smart contract from which 13 test cases are pass and 0 test cases fails as shown below.

**Total Test cases = 13**

**Pass Test cases = 13**

**Fail Test cases = 0**

This shows our test cases are more detailed and are capable of reporting faults in a more robust manner.

## 4.2. Test Cases Mapping

We mapped all our written test cases for both case studies against all aspects proposed in our model as shown in Table-9.

**Table 9:** Test Cases Mapping

Case Study No.	Test Case No.	Pre-Conditions			Post-Conditions			Contract Invariant Classes			Roles		
		Account with Sufficient Balance	Address	Inputs	Transaction	Current State	Event	State	Address	Inputs	Account with Sufficient Balance	Inputs	Role
1	1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	2	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	3	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	4	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
	5	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	6	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	7	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	8	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	9	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	11	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	12	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	13	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
	14	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	15	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	16	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
	17	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
	18	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	19	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	20	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
	21	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

22	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
23	x	✓	✓	x	x	x	x	x	x	x	✓	x
24	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
25	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
26	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
27	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
28	x	✓	✓	x	x	x	x	x	x	x	✓	x
29	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
30	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
31	x	✓	✓	x	x	x	x	x	x	x	✓	x
32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
33	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
34	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
35	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
36	x	✓	✓	x	x	x	x	x	x	x	✓	x
37	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
38	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
39	x	✓	✓	x	x	x	x	x	x	x	✓	✓
40	x	✓	✓	x	x	x	x	x	x	x	✓	x
41	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
42	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
43	x	✓	✓	x	x	x	x	x	x	x	✓	x
44	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
45	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
46	x	✓	✓	x	x	x	x	x	x	x	✓	x
47	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
48	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
49	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
50	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
51	x	✓	✓	x	x	x	x	x	x	x	✓	x
52	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
53	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
54	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
55	x	✓	✓	x	x	x	x	x	x	x	✓	x
56	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
57	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
58	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
59	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
60	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
61	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓
62	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

	63	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	64	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	65	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	66	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	67	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	68	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	69	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	70	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	71	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	71	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2	1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	2	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	3	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	4	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	6	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	7	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
	8	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
	9	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
	10	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓
	11	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	12	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓
	13	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓

### 4.3. Comparison of techniques

In this section we explain three different testing techniques already in practice for the testing of smart contracts and then for comparison we explain our testing technique.

sFuzz Technique	Our Approach
An automated tool used for the testing of Ethereum virtual machine (EVM) smart contracts proposed by Nguyen et al [66] . As we know that the byte code of the smart contract is executes on EVM instead of direct smart contract code, so for this tool the byte code of smart contract is required for testing. This tool is validating and detecting security vulnerabilities in EVM smart contracts. This tool is generating test cases and security	Our model-based testing is not for the EVM based smart contracts. Before converting smart contracts into bytecode, we propose a model for testing smart contracts from different aspects. We extract test cases from different aspects of smart contract. We also provide a matrix traceability to help testers to attain maximum testing coverage.

vulnerabilities on fuzzy smart contracts. For generating test cases, this tool takes time.	
--	--

<b>Mutation Testing</b>	<b>Our Approach</b>
Wu et al. proposed a mutating based testing for smart contracts [67]. They proposed different mutation operators for Ethereum smart contracts mutation testing. By using it, they write test cases for validating and checking security vulnerabilities for smart contracts.	We propose a model for testing smart contracts from different aspects. We extract test cases from different aspects of smart contract. We also provide a matrix traceability to help testers to attain maximum testing coverage.

<b>Fuse Testing</b>	<b>Our Approach</b>
Chan et al. propose the progress of new testing service named fuse that is supporting fuzzy testing of smart contracts. In their approach, they used seven different classes of vulnerabilities and defined a set of test oracle. They have proposed a fuzzy testing with constant seeding to generate test transactions and evaluate fuzz-tests.	We propose a model for testing smart contracts from different aspects. We extract test cases from different aspects of smart contract. We also provide a matrix traceability to help testers to attain maximum testing coverage.

#### 4.4. Generalization of our proposal

It is important to highlight that contract has state, address, variables value, functions, transactions, events, inputs and outputs. Contract invariant is an essential aspect. Roles aspect may vary in different proposals. We, therefore, consider the following three aspects for a generalized technique

- Pre-Condition
- Post-Condition
- Contract Invariants Classes

## 4.5. Validation

On the bases of information provided in model we write test cases manually but without applying any coverage formal notation. By using fault seeding we evaluate the completeness of our test cases. The test set capability to detect or address similar errors in a system i.e., a measure of confidence in our test suite [69] is a statistical measure provided by the seeded faults percentage. We decide which fault to be introduced in system we developed different rules and find different fault types for automatically seeding them. After applying all rules on system code, we execute test suite to assess its quality.

In order to decide which faults to introduce we identified suitable fault types, and then developed rules for seeding them automatically. After applying the rules to the code of the system, we execute the entire test suite to assess its quality. We added test cases in an iterative process until all the seeded faults were identified. We seed a fault in smart contract by changing the value of UPC and productID as shown in figure 27.

```
// Define few constants and assign a few sample accounts to
var sku = 1
var upc = 1
var ownerID = accounts[0]
const originFarmerID = accounts[1]
const originFarmName = "Ali Ahmmad"
const originFarmInformation = "Rawalpindi"
const originFarmLatitude = "-38.239770"
const originFarmLongitude = "144.341490"
var productID = upc + sku

var sku = 1
var upc = 3
var ownerID = accounts[0]
const originFarmerID = accounts[1]
const originFarmName = "Ali Ahmmad"
const originFarmInformation = "Rawalpindi"
const originFarmLatitude = "-38.239770"
const originFarmLongitude = "144.341490"
var productID = upc - sku
const productNotes = "Best beans for Espresso"
```

**Figure 27:** Fault Seeding in Smart contract

Faults are classified into two types; domain and computational faults by [70]. The result of control flow error is a domain fault in which program executed in a wrong path while when a program executed in a correct path but give incorrect results is a computation fault. More specifically, we have followed the fault types discussed in [71], which also supports calculating a measure of

confidence in a test suite. Fault types are classified by [71], into wrong declaration, wrong assignment, wrong procedure handling, control faults and I/O faults. Considering Case study No. 1, we consider 71 test cases for validation purposes. We seed 41 faults and run our test cases. We rerun our test cases and find all faults giving a confidence of 99%. Considering Case study No.2, we consider 13 test cases for validation purposes. We seed 7 faults and run our test cases. We rerun our test cases and find all faults giving a confidence of 99%. Our results are shows in Table-9.

***Table 9: Results***

<b>Case Study</b>	<b>Total Test cases</b>	<b>Faults Seeding</b>	<b>Faults Found</b>
1.	71	41	41
2.	13	7	7

We can say that if faults will seed or not in to smart contracts, our technique will test all aspects of smart contract.

## Chapter 5

# Conclusion and Future Work

Ethereum blockchain is more popular between Finance related organizations. Ethereum allows many other features other than the transfer of cryptocurrency. Smart contracts on Ethereum run independently, they will trigger or run when an external application or another contract can call them. Smart contracts in Ethereum written in Solidity Language. Once the smart contract is deployed on the network then it will never be changeable you have to write a new contract with fix. Due to this reason smart contracts testing is much important than any other thing. If anyone wants to interact with the Ethereum then their accounts must be created on Ethereum Blockchain by using Metamask. Metamask is a browser extension which will be added after installation. On Ethereum for doing transaction a gas fee is charged as a processing fee. Whenever a transaction is processed miners on the blockchain will mine the transaction and in return a processing fee will be charged to the sender. Gas fee will must be paid by the sender with the transaction amount. On Ethereum if you deploy any smart contract, on the basis of complexity of the smart contract gas fee will be charged which will be paid by the owner. After all these observations we can say that for doing transaction or smart contract deployment one must have account balance.

In this thesis we worked on the testing of smart contracts as we didn't find any research on model-based testing of smart contracts. We propose a model-based technique for testing smart contracts. We identify different aspects of smart contract to write test cases. We also provide a testing traceability matrix which shows the maximum test coverage. We implement our model on two case studies for testing smart contracts. We also evaluate our test suite capability by seeding faults in our system. In an iterative process we added test cases until all seeded faults were identified. For the testing and deployment of smart contracts we used Truffle, Ganache, Node.js and Git. We used Metamask and RemixIDE for learning the interaction of smart contracts with applications. At the end we conclude that to avoid smart contract failure issue we have to test all pre-conditions, post-conditions and invariants of smart contracts.

Private Blockchain is a blockchain in which a centre authority is there and if anyone wants to use the private blockchain then it must be authenticate by the central authority while the public blockchain is completely opposite of it. Ethereum is a public blockchain, anyone can use it by



creating account. Some companies used this blockchain and develop their own private blockchains i.e. Amazon build a private blockchain based on Ethereum and now they are offering private blockchain services with smart contracts. Some other companies develop their own games using Ethereum Blockchain in which on the start of the game they asked for Ethereum account to play the game. As we know Ethereum is a public blockchain, different companies' uses it for their own different purposes. The future work will be that a general programming language can be proposed to write smart contracts for any case study.

## REFERENCES

1. A. Savelyev. "Contract law 2.0: 'Smart' contracts as the beginning of the end of classic contract law". Information & Communications Technology Law 26.2 (2017).
2. M. Gates. Blockchain: Ultimate guide to understanding blockchain, bitcoin, cryptocurrencies, smart contracts and the future of money. CreateSpace Independent Publishing Platform, 2017.
3. Szabo, Nick. "Formalizing and securing relationships on public networks." First Monday 2.9 (1997).
4. Bahga, A. and Madiseti, V.K. (2016) Blockchain Platform for Industrial Internet of Things. Journal of Software Engineering and Applications, 9, 533-546. <http://dx.doi.org/10.4236/jsea.2016.910036>
5. Gerhold, M., Stoelinga, M. Model-based testing of probabilistic systems. Form Asp Comp 30, 77–106 (2018). <https://doi.org/10.1007/s00165-017-0440-4>
6. Frantz, Christopher K., and Mariusz Nowostawski. "From institutions to code: Towards automated generation of smart contracts." 2016 IEEE 1st International Workshops on Foundations and Applications of Self\* Systems (FAS\* W). IEEE, 2016.
7. N. Szabo. Smart Contracts. [Online]. Available: <http://szabo.best.vwh.net/smart.contracts.html>
8. N. Szabo. The Idea of Smart Contracts. [Online]. Available: [http://szabo.best.vwh.net/smart\\_contracts\\_idea.html](http://szabo.best.vwh.net/smart_contracts_idea.html)
9. Mense, Alexander & Flatscher, Markus. (2018). Security Vulnerabilities in Ethereum Smart Contracts. 375-380. 10.1145/3282373.3282419.
10. Khan T.A., Runge O., Heckel R. Testing against Visual Contracts: Model-Based Coverage. In: Ehrig H., Engels G., Kreowski HJ., Rozenberg G. (eds) Graph Transformations. ICGT. Lecture Notes in Computer Science, vol 7562. Springer, Berlin, Heidelberg
11. Heckel, Reiko & Khan, Tamim & Machado, Rodrigo. Towards Test Coverage Criteria for Visual Contracts. ECEASST. 41. 10.14279/tuj.eceasst.41.667.
12. Runge, Olga, Tamim Ahmed Khan, and Reiko Heckel. "Test case generation using visual contracts." Electronic Communications of the EASST 58.
13. Dika, Ardit, and Mariusz Nowostawski. "Security Vulnerabilities in Ethereum Smart Contracts." 2018 IEEE International Conference on Internet of Things (iThings) and IEEE

- Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, 2018.
14. Wang, Shuai, Chengyu Zhang, and Zhendong Su. "Detecting nondeterministic payment bugs in Ethereum smart contracts." *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019): 189.
  15. Li, Zixin, et al. "MuSC: A Tool for mutation testing of Ethereum smart contract." 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019.
  16. Chan, W. K., and Bo Jiang. "Fuse: An Architecture for Smart Contract Fuzz Testing Service." 2018 25th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2018.
  17. Zhang, William, et al. "MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract." *arXiv preprint arXiv:1911.00570* (2019).
  18. Parizi, Reza M., et al. "Empirical vulnerability analysis of automated smart contracts security testing on blockchains." *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 2018.
  19. Fu, Ying, et al. "Evmfuzz: Differential fuzz testing of ethereum virtual machine." *arXiv preprint arXiv:1903.08483* (2019).
  20. Hasan, Haya R., and Khaled Salah. "Combating deepfake videos using blockchain and smart contracts." *Ieee Access* 7 (2019): 41596-41606.
  21. Destefanis, Giuseppe, et al. "Smart contracts vulnerabilities: a call for blockchain software engineering?." 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, 2018.
  22. Swan, Melanie. *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.
  23. Zhao, J. Leon, Shaokun Fan, and Jiaqi Yan. "Overview of business innovations and research opportunities in blockchain and introduction to the special issue." (2016): 1-7.
  24. Bjørnstad, Magnus Vitsø, Simen Krogh, and Joar Gunnarsjaa Harkestad. *A study on blockchain technology as a resource for competitive advantage*. MS thesis. NTNU, 2017.
  25. Seebacher, Stefan, and Ronny Schüritz. "Blockchain technology as an enabler of service systems: A structured literature review." *International Conference on Exploring Services Science*. Springer, Cham, 2017.

26. Zhao, J. Leon, Shaokun Fan, and Jiaqi Yan. "Overview of business innovations and research opportunities in blockchain and introduction to the special issue." (2016): 1-7.
27. Seebacher, Stefan, and Ronny Schüritz. "Blockchain technology as an enabler of service systems: A structured literature review." *International Conference on Exploring Services Science*. Springer, Cham, 2017.
28. J. L. Zhao, S. Fan, and J. Yan. "Overview of business innovations and research opportunities in blockchain and introduction to the special issue". Springer (2016).
29. R. Böhme et al. "Bitcoin: Economics, technology, and governance". *Journal of Economic Perspectives* 29.2 (2015).
30. J. Garay, A. Kiayias, and N. Leonardos. "The bitcoin backbone protocol: Analysis and applications". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015.
31. Y. Lewenberg, Y. Sompolinsky, and A. Zohar. "Inclusive block chain protocols". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2015.
32. H. Wang, K. Chen, and D. Xu. "A maturity model for blockchain adoption". *Financial Innovation* 2.1 (2016).
33. F. Tschorsch and B. Scheuermann. "Bitcoin and beyond: A technical survey on decentralized digital currencies". *IEEE Communications Surveys & Tutorials* 18.3 (2016).
34. S. Ølnes. "Beyond bitcoin enabling smart government using blockchain technology". In: *International Conference on Electronic Government and the Information Systems Perspective*. Springer. (2016).
35. D. Lee Kuo Chuen, Ed., *Handbook of Digital Currency*, 1st ed. Elsevier, 2015. [Online]. Available: <http://EconPapers.repec.org/RePEc:eee:monogr:9780128021170>
36. V. Buterin, "A next-generation smart contract and decentralized application platform," white paper.
37. Vujičić, Dejan, Dijana Jagodić, and Siniša Randić. "Blockchain technology, bitcoin, and Ethereum: A brief overview." *2018 17th international symposium infoteh-jahorina (infoteh)*. IEEE, 2018.
38. Lu, Yang. "Blockchain: A survey on functions, applications and open issues." *Journal of Industrial Integration and Management* 3.04 (2018): 1850015.

39. Merkle, Ralph C. "A digital signature based on a conventional encryption function." Conference on the theory and application of cryptographic techniques. Springer, Berlin, Heidelberg, 1987.
40. Merkle, Ralph C. "Protocols for public key cryptosystems." 1980 IEEE Symposium on Security and Privacy. IEEE, 1980.
41. Bergquist, Jonatan. "Blockchain Technology and Smart Contracts: Privacy-Preserving Tools." (2017).
42. Wüst, Karl, and Arthur Gervais. "Do you need a blockchain?." 2018 Crypto Valley Conference on Blockchain Technology (CVCBT). IEEE, 2018.
43. Vokerla, Rahul Rao, et al. "An Overview of Blockchain Applications and Attacks." 2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN). IEEE, 2019.
44. Kikitamara, Sesaria, M. C. J. D. van Eekelen, and Dipl Ing Jan-Peter Doomernik. "Digital identity management on blockchain for open model energy system." Unpublished Masters thesis–Information Science (2017).
45. MultiChainTeam. MultiChain Official Website. (2017). url: <https://www.multichain.com/>. (visited on 06/10/2020).
46. MONAX-Team. MONAX: Deploy smart, legally-binding contracts on the blockchain. (2016). url: <https://monax.io/>. (visited on 06/10/2020).
47. M. Iansiti and K. R. Lakhani. The Truth About Blockchain. Harvard Business Review(). URL: <https://hbr.org/2017/01/the-truth-about-blockchain> (visited on 06/10/2020).
48. E. Community. A Next Generation Blockchain. URL: <https://ethdocs.org/en/latest/introduction/what-is-ethereum.html#a-next-generation-blockchain> (visited on 06/10/2020).
49. I. Bashir. Mastering Blockchain. English. 1st ed. GB: Packt Publishing, 2017. ISBN:9781787125445.URL:<https://ebookcentral.proquest.com/lib/tut/detail.action?docID=4826445>.
50. Wood, Gavin. "Ethereum: A secure decentralised generalised transaction ledger." Ethereum project yellow paper 151.2014 (2014): 1-32.
51. E. C. Organization. Ethereum Classic Website. URL: <https://ethereumclassic.org> (visited on 06/11/2020).
52. E. Organization. Ethereum Website. URL: <https://ethereum.org> (visited on 06/11/2020).

53. Tikhomirov, Sergei. "Ethereum: state of knowledge and research perspectives." International Symposium on Foundations and Practice of Security. Springer, Cham, 2017.
54. Tran, Hung. "Enabling a decentralized organization through smart contracts and tokens on the Ethereum blockchain." (2018).
55. Advisors, E. T. H., et al. "ETHEREUM ANALYTICS." (2019).
56. Sillaber, Christian, and Bernhard Wautl. "Life cycle of smart contracts in blockchain ecosystems." Datenschutz und Datensicherheit-DuD 41.8 (2017): 497-500.
57. Khan, Tamim Ahmed, and Reiko Heckel. "A methodology for model-based regression testing of web services." 2009 Testing: Academic and Industrial Conference-Practice and Research Techniques. IEEE, 2009.
58. Mohanta, Bhabendu Kumar, Soumyashree S. Panda, and Debasish Jena. "An overview of smart contract and use cases in blockchain technology." 2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT). IEEE, 2018.
59. Wohrer, Maximilian, and Uwe Zdun. "Smart contracts: security patterns in the ethereum ecosystem and solidity." 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, 2018.
60. V. Buterin: A next-generation smart contract and decentralized application platform. White paper, 2014
61. Ethereum, Documentation. URL:<http://www.ethdocs.org/en/latest/>, Last visited June 19, 2020
62. Delmolino, Kevin, et al. "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab." International conference on financial cryptography and data security. Springer, Berlin, Heidelberg, 2016.
63. Stack Exchange URL:<https://ethereum.stackexchange.com/questions/2404/upgradeable-smart-contract> , Last visited June 19, 2020
64. ZeppelinOS Documentation URL:<https://docs.zepelinos.org/docs/1.0.0/start> , Last visited June 19, 2020
65. Khan, Tamim Ahmed, and Reiko Heckel. "On model-based regression testing of web-services using dependency analysis of visual contracts." International Conference on Fundamental Approaches to Software Engineering. Springer, Berlin, Heidelberg, 2011.

66. Nguyen, Tai D., et al. "sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts." arXiv preprint arXiv:2004.08563 (2020).
67. Wu, Haoran, et al. "Mutation testing for ethereum smart contract." arXiv preprint arXiv:1908.03707 (2019).
68. Chan, Wing Kwong, and Bo Jiang. "Fuse: An architecture for smart contract fuzz testing service." 2018 25th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2018.
69. Pfleeger, S.L.: Software Engineering: Theory and Practice. Prentice Hall PTR, Upper Saddle River (2001)
70. Howden, W.: Reliability of the path analysis testing strategy. IEEE Transactions on Software Engineering SE-2(3), 208–215 (1976)
71. Pasquini, A., Agostino, E.D.: Fault seeding for software reliability model validation. Control Engineering Practice 3(7), 993–999 (1995)
72. Wüstholtz, Valentin, and Maria Christakis. "Harvey: A greybox fuzzer for smart contracts." arXiv preprint arXiv:1905.06944 (2019).
73. Hartel, Pieter, and Richard Schumi. "Mutation Testing of Smart Contracts at Scale." International Conference on Tests and Proofs. Springer, Cham, 2020.
74. Andesta, Erfan, Fathiyeh Faghieh, and Mahdi Fooladgar. "Testing Smart Contracts Gets Smarter." arXiv preprint arXiv:1912.04780 (2019).
75. Ashraf, Imran, et al. "GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities." IEEE Access (2020).
76. Chapman, Patrick, et al. "Deviant: A Mutation Testing Tool for Solidity Smart Contracts." 2019 IEEE International Conference on Blockchain (Blockchain). IEEE, 2019.
77. Akca, Sefa, Ajitha Rajan, and Chao Peng. "SolAnalyser: A Framework for Analysing and Testing Smart Contracts." 2019 26th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2019.
78. Nehai, Zeinab, Pierre-Yves Piriou, and Frederic Daumas. "Model-checking of smart contracts." 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, 2018.
79. Ellul, Joshua, and Gordon J. Pace. "Runtime verification of ethereum smart contracts." 2018 14th European Dependable Computing Conference (EDCC). IEEE, 2018.

## APPENDIX A

S No	Test Case	Pre-Conditions	Post-Condition	Output	Expected Results	Actual Results(Pass/Fail)
1.	Add Product Successfully	Product must have a name and price	Product should be add in to the catalogue.	Product added successfully.	Product added successfully.	Pass
2.	Product not added without name and price	Product must have name and price	Product should not be add in to catalogue without name and price.	No Product added in catalogue without name and price.	Product not add in catalogue without name and price.	Pass
3.	Product not added without name	Product must have name	Product should not be add in to catalogue without name.	No Product added in catalogue without name.	Product not add in catalogue without name.	Pass
4.	Product not added without price	Product must have price	Product should not be add in to catalogue without price.	No Product added in catalogue without price.	Product not add in catalogue without price.	Pass
5.	Sale Product Transaction Successfully	Product must have an id	After Successful transaction product should be remove from catalogue.	Transaction done successfully and removed from catalogue.	Transaction done successfully and removed from catalogue.	Pass
6.	Trying to buy same product again	Product must have an id	Product should not be sale again.	Transaction Failed	Transaction Fail	Pass
7.	Trying to add	Account informatio	Product should not	No Product	Product should not be add in to	Pass



	product without account	n required for adding product detail	be add in to catalogue without account.	added in catalogue	catalogue without account.	
<b>8.</b>	Trying to buy product without account	Buyer must have an account	Product should not be sale.	Transaction failed	Transaction Fail	<b>Pass</b>
<b>9.</b>	Trying to buy product with insufficient balance	Buyer must have an account with sufficient balance	Product should not be sale.	Transaction failed	Transaction Fail	<b>Pass</b>
<b>10.</b>	Trying to add product with insufficient balance	Seller must have an account with sufficient balance	Product should not be add.	Product not added in catalogue.	Product should not be add in catalogue	<b>Pass</b>
<b>11.</b>	Add multiple products in to the catalogue	Seller must have an account with sufficient balance	Products should be add	All products added successfully	Products should be add	<b>Pass</b>
<b>12.</b>	Trying to buy product with invalid product id	Buyer must have an account with sufficient balance	Product should not be sale.	Transaction failed	Transaction Fail	<b>Pass</b>
<b>13.</b>	Trying to add product with zero price	Seller must have an account with sufficient balance	Product should not be add	Product not added in catalogue	Product should not be add in catalogue	<b>Pass</b>