Model-Driven Approach to Smart Contract Development With Automatic Code Generation

A Thesis

presented to

the Faculty of Natural and Applied Sciences

at Notre Dame University-Louaize

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Christian Eid

November 2020
Notre Dame University - Louaize
Faculty of Natural and Applied Sciences
Department of Computer Science

We hereby approve the thesis of

Christian Eid

Candidate for the degree of Master of Science in Computer Science

Dr. Pierre Akiki
Supervisor, Chair

Dr. Marie Khair
Committee Member

Dr. Nazir Hawi
Committee Member
Declaration

I hereby declare that I am the sole author of this thesis. To the best of my knowledge, this thesis contains no material previously published by any other person except where due acknowledgment has been made.
Acknowledgments

I wish to thank Dr. Pierre Akiki for his valuable insight and guidance throughout my academic years, especially with his role as my thesis advisor.

I also wish to thank Dr. Hoda Maalouf for her assistance and for serving as my advisor during my master’s program.

I would also like to thank my wife and my parents who have supported me throughout the entire process.

A special word of thanks goes to the Faculty of Natural and Applied Sciences department at Notre Dame University - Louaize for the effort and dedication it has invested in us to provide us with the knowledge and background necessary to lead a successful career.
Table of Contents

Acknowledgments ........................................................................................................ v
Table of Contents .......................................................................................................... vi
List of Figures ................................................................................................................ viii
List of Tables .................................................................................................................. x
List of Code Listings ...................................................................................................... xi
List of Abbreviations .................................................................................................... xii
Abstract .......................................................................................................................... xiv

Chapter 1: Introduction and Problem Definition ......................................................... 1

1.1 Introduction to the General Problem ....................................................................... 1
1.2 Problem Definition .................................................................................................. 1
1.3 Research Objectives .............................................................................................. 2
1.4 Approach and Main Results .................................................................................. 2
1.5 Thesis Organization ............................................................................................... 2

Chapter 2: Background and Motivation ........................................................................ 3

2.1 Definition of the Basic Concepts ........................................................................... 3
  2.1.1 Blockchain Overview ...................................................................................... 4
2.2 Blockchain Architecture ....................................................................................... 8
  2.2.1 Block .............................................................................................................. 9
  2.2.2 Cryptography .................................................................................................. 10
2.3 Consensus Algorithms ......................................................................................... 11
  2.3.1 Proof of Work (PoW) .................................................................................... 12
  2.3.2 Proof of Stake (PoS) .................................................................................... 13
  2.3.3 Practical Byzantine Fault Tolerance (PBFT) .................................................. 14
2.4 Smart Contracts ..................................................................................................... 15
  2.4.1 Ethereum ....................................................................................................... 17
2.5 Model-Driven Engineering .................................................................................... 18
  2.5.1 Model-Driven Architecture .......................................................................... 21
2.6 Previous Work in the Subject ................................................................................ 22
  2.6.1 Agent-Based Approach ................................................................................ 23
  2.6.2 Business Process-Based Approach ................................................................. 23
  2.6.3 State Machine Approach ................................................................................ 25
  2.6.4 UML Approach ............................................................................................ 27
2.7 Research Motivation .............................................................................................. 28
Chapter 3: Model-Driven Smart Contracts ............................................ 29

3.1 Introduction ..................................................................................... 29
3.2 Smart Contract Development Language ........................................ 30
3.3 Smart Contract Generation Tool ...................................................... 37
  3.3.1 XML Validation ........................................................................ 39
  3.3.2 Code Generation ....................................................................... 41
3.4 Evaluation ....................................................................................... 52
  3.4.1 Real-World Case ...................................................................... 52
  3.4.2 Efficiency, Scalability, and Line of Code Comparison ................. 61
3.5 Threats to Validity and Limitations ................................................ 65

Chapter 4: Conclusion ........................................................................ 66

4.1 Summary of the Main Results ......................................................... 66
4.2 Main Contributions of the Thesis .................................................... 66
4.3 Possible Extensions and Future Work .............................................. 67

Bibliography ....................................................................................... 68

Appendix A: Open Auction Generated Code ....................................... 78
List of Figures

Figure 2.1 Top 3 Cryptocurrencies By Value (*Cryptocurrency Market Capitalizations*, n.d.). 4

Figure 2.2: Public Blockchain ........................................................................................................... 6

Figure 2.3: Private Blockchain ........................................................................................................ 7

Figure 2.4: Consortium Blockchain ................................................................................................. 8

Figure 2.5: Continuous Sequence of Blocks in a Blockchain (Zheng et al., 2017) ....................... 9

Figure 2.6: Block Structure (Zheng et al., 2017) .......................................................................... 10

Figure 2.7: Fork Scenario (Wang et al., 2018) .............................................................................. 13

Figure 2.8: PBFT Phases (Castro & Liskov, 1999) ......................................................................... 14

Figure 2.9: Cryptocurrency System With Smart Contract (Delmolino et al., 2016) ................. 16

Figure 2.10: Modeling Spectrum (Beydeda et al., 2005) ............................................................... 19

Figure 2.11: Relationship Between Acronyms of Model-Driven Software (Smith et al., 2015) ........................................................................................................................................... 20

Figure 2.12: MDA Principles (Brambilla et al., 2017) .................................................................. 21

Figure 3.1 XML-Based Language Class Diagram ............................................................................ 31

Figure 3.2 Complete Code Generation Activity Diagram ............................................................... 38

Figure 3.3 XML-based Language Validation .................................................................................. 39

Figure 3.4 Code Generation Workflow ........................................................................................... 41

Figure 3.5 Home Page UI .................................................................................................................. 50

Figure 3.6 Blockchain Selection UI .................................................................................................. 51

Figure 3.7 Open Auction Activity Diagram ...................................................................................... 57
Figure 3.8 Generation Time vs Model Number of Lines (1.87 GHz) .......................... 62
Figure 3.9 Generation Time vs Model Number of Lines (3.29 GHz) .......................... 62
Figure 3.10 Number of Lines of Code Needed for the Same Contract on Different Platform 64
List of Tables

Table 2.1 Consensus Algorithm Comparison ................................................................. 15
Table 2.2 Summary and Comparison – Business Process Approach .................................. 25
Table 2.3 Summary and Comparison – Finite State Machine Approach ............................. 26
Table 2.4 Summary and Comparison – Other Approaches ............................................... 27
Table 3.1 Accepted Elements .......................................................................................... 36
Table 3.2 Accepted Attributes ....................................................................................... 37
Table 3.3 Summary of Tests Timing .............................................................................. 63
# List of Code Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Smart Contract Sample in XML</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Accepted Arguments in Elements, Dictionary example</td>
<td>40</td>
</tr>
<tr>
<td>3.3</td>
<td>GetReservedDataMapping Solidity Method</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Solidity Event Generation</td>
<td>44</td>
</tr>
<tr>
<td>3.5</td>
<td>Generated Solidity Code</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>GetReservedDataMapping Neo C# Method</td>
<td>47</td>
</tr>
<tr>
<td>3.7</td>
<td>Neo Event Generation</td>
<td>48</td>
</tr>
<tr>
<td>3.8</td>
<td>Generated C# Neo Code</td>
<td>48</td>
</tr>
<tr>
<td>3.9</td>
<td>Cryptocurrency Implementation</td>
<td>52</td>
</tr>
<tr>
<td>3.10</td>
<td>Solidity Cryptocurrency Contract</td>
<td>53</td>
</tr>
<tr>
<td>3.11</td>
<td>C# Cryptocurrency Contract</td>
<td>54</td>
</tr>
<tr>
<td>3.12</td>
<td>Open Auction XML</td>
<td>57</td>
</tr>
<tr>
<td>3.13</td>
<td>Open Auction Solidity Partial Contract</td>
<td>59</td>
</tr>
<tr>
<td>3.14</td>
<td>Open Auction Neo C# Partial Contract</td>
<td>59</td>
</tr>
<tr>
<td>A.1</td>
<td>Open Auction Solidity Code Generated</td>
<td>78</td>
</tr>
<tr>
<td>A.2</td>
<td>Open Auction C# Code Generated</td>
<td>80</td>
</tr>
</tbody>
</table>
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABM</td>
<td>Agent-Based Models</td>
</tr>
<tr>
<td>ADICO</td>
<td>Attributes, Deontic, Aim, Conditions and Or else</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Model and Notation</td>
</tr>
<tr>
<td>CIM</td>
<td>Computation Independent Model</td>
</tr>
<tr>
<td>dApps</td>
<td>Decentralized Applications</td>
</tr>
<tr>
<td>EVM</td>
<td>Ethereum Virtual Machine</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>LLL</td>
<td>Low-level Lisp-like Language</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
</tr>
<tr>
<td>NeoVM</td>
<td>Neo Virtual Machine</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>PBFT</td>
<td>Practical Byzantine Fault Tolerance</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PoS</td>
<td>Proof of Stake</td>
</tr>
<tr>
<td>PoW</td>
<td>Proof of Work</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>SHA</td>
<td>Secure Hash Algorithm</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Abstract

Existing smart contract development languages are unique to their related blockchain platforms. Thus, multiple versions of the same smart contract must be developed for different blockchains.

This thesis presents a model-driven approach to smart contract development with automatic code generation. This approach encompasses a new XML-based programming language to develop smart contracts and a code generator that transforms those contracts to a platform-specific programming language.

The XML-based language was discussed, and all supported constructs were illustrated. The code generation tool currently supports Ethereum and Neo smart contracts that are written in Solidity and C# respectively. The language and its supporting code generation tool were tested with two real-world smart contract examples. The properties and capabilities of the language and tool were demonstrated by generating Solidity and C# code correctly and efficiently. In comparison to Solidity and C#, the new language required fewer lines of code to represent the same smart contracts. Another advantage of this language is that developers can learn one language instead of several which reduces the learning curve and development time.

Keywords: Blockchain, Smart Contracts, Model-Driven, Code Generation, XML, Ethereum, NEO, Solidity, C#
Chapter 1: Introduction and Problem Definition

This chapter presents the problem definition, research objectives, main result, and the thesis organization.

1.1 Introduction to the General Problem

The emergence of blockchain technology in recent years with smart contracts has opened the door for the development of decentralized applications. Blockchain platforms are varying and new chains are becoming popular. This behavior increases the difficulty of blockchain development because each blockchain supports different smart contracts and programming languages. New blockchain developers also face the challenge of selecting the blockchain platform that will suit them best.

1.2 Problem Definition

Smart contract development is done on a platform basis because each blockchain has a programming language that supports it, which reduces the ability for code reusability and modularity. In case a smart contract needs to be developed and deployed on different blockchains, the developer will need to know both languages and will have to develop each contract separately. This task is tricky, complex, and time-consuming. There is also a gap in the existing research regarding this subject with no definitive solution or framework to use.
1.3 Research Objectives

The main purpose of this research is to facilitate the development of smart contracts on different blockchain platforms, by providing a tool that eliminates the need to re-code the whole contact in each blockchain specific language. Also, this research aims to reduce the development time, development complexity, and the learning curve needed for new blockchain developers.

1.4 Approach and Main Results

A new XML-based programming language is developed with a compatible code generator to transform the XML file into a working smart contract source code supporting Ethereum and Neo blockchains. The XML structure is simple, and the percentage difference between the lines of code in XML needed was 60.86\% compared to Solidity and 133.05\% to Neo’s C#. The code converter is efficient and capable of generating 15,000 lines of the XML-based language into the targeted platform on average in 2 seconds.

1.5 Thesis Organization

Chapter 2 describes the basic concepts of blockchains by visiting the blockchain architecture, consensus algorithms, and smart contracts. Then, model-driven engineering is discussed and is followed by a literature review section that shows and compares recent studies related to this research. In Chapter 3, the original work is presented in detail alongside two use case scenarios that evaluates the proposed approach. The research is concluded in Chapter 4 by listing the main contribution and possible future work.
Chapter 2: Background and Motivation

This chapter starts by presenting the basic concepts of blockchain technology and then provides a more in-depth description of smart contracts and related notions. The related work is discussed and compared, and the thesis motivation is expressed.

2.1 Definition of the Basic Concepts

Blockchain technology has been growing in the last decade. It all started in 2008 when a white paper was published by a pseudonymous person or persons named Satoshi Nakamoto. It introduced bitcoin as an electronic coin that can be transferred directly between peers in a trustless medium while keeping a public history of all transactions (Nakamoto, 2008). Later, the technology that bitcoin is built upon was named blockchain. Some of the key features of blockchain are its decentralized nature, the anonymity of data, persistence, and auditability. Even though the well-known applications of blockchain are cryptocurrencies, the range of applications where it can be used are limitless (Wang et al., 2018). For example, in the healthcare field, medical records can be shared on the blockchain and owned by the patient instead of having them distributed partially on different systems (Yue et al., 2016). In the internet of things (IoT) applications, blockchain is very promising as it improves security and data obscurity of the interconnected devices, and increases maintainability (Abou Jaoude & Saade, 2019).
Chapter 2: Background and Motivation

2.1.1 Blockchain Overview

Blockchain can be described as an immutable public ledger or a cryptographically secured distributed database for all transactions made between participants (Crosby, 2016). All information written on the block cannot be removed or altered. Three versions of blockchain are identified, version 1 involves the usage of blockchain to build cryptocurrencies, version 2 introduces smart contracts, and version 3 introduces many kinds of decentralized apps (dApp) (Swan, 2015). These versions will be introduced gradually in this thesis.

2.1.1.1 Cryptocurrencies

Trust is an important subject when dealing with digital transactions between parties online, especially in finance. If Bob wants to buy a product directly from Alice, there is no guarantee that once the payment is received by Alice, the product will be sent to Bob as agreed. Hence, a third-party financial institution is used to process and keep track of the transactions; this institution is trusted by both parties doing the operation (Crosby, 2016). On the other hand, cryptocurrencies are digital currencies that can be transferred between peers directly. It is decentralized and is not issued by a central authority like regular currencies. The first cryptocurrency ever created is bitcoin and is still the leading digital cash (Corbet et al., 2019). The current top 3 cryptocurrencies by market capitalization are shown in Figure 2.1.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Market Cap</th>
<th>Price</th>
<th>Volume (24h)</th>
<th>Price Graph (7d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bitcoin</td>
<td>$190,333,038,757</td>
<td>$10,296.61</td>
<td>$50,278,062,471</td>
<td><img src="https://example.com" alt="Price Graph" /></td>
</tr>
<tr>
<td>2</td>
<td>Ethereum</td>
<td>$41,119,910,931</td>
<td>$385.37</td>
<td>$31,465,867,973</td>
<td><img src="https://example.com" alt="Price Graph" /></td>
</tr>
<tr>
<td>3</td>
<td>Tether</td>
<td>$14,444,517,352</td>
<td>$1.00</td>
<td>$56,221,702,638</td>
<td><img src="https://example.com" alt="Price Graph" /></td>
</tr>
</tbody>
</table>

Figure 2.1 Top 3 Cryptocurrencies By Value (Cryptocurrency Market Capitalizations, n.d.)
Cryptography and consensus algorithms are used to provide trust for the parties doing a transaction without any intermediary (Crosby, 2016). In bitcoin, every user has their copy of all the ledgers (blocks) and every transaction made on the bitcoin blockchain will propagate to all the users. Thus, the double pending problem can occur due to this structure alone, where users can cheat by sending the payment multiple times to different users. As a preventive measure, bitcoin uses the Proof of Work (PoW) consensus discussed in Section 2.3.1, which makes the users perform mathematical calculations to solve a cryptographic puzzle (Tschorsch & Scheuermann, 2016). Once the puzzle is solved, a block is added to the blockchain. This process is called “mining”. In precise, a nonce value should be discovered by the miner (bitcoin user) that will be hashed with other fields to generate a new value below a given threshold. The new block will be added to the blockchain in case that nonce is found. A predefined amount of bitcoins will be given to the user who solved this problem (Gervais et al., 2014).

2.1.1.2 Types of Blockchain

The main types of blockchains are Public and Private. The Hybrid and Consortium blockchains types are variations of the main types. All types share the following features (“Types of Blockchains,” 2019):

- Contain a cluster of nodes
- Work on peer-to-peer networks
- Update a copy of the shared ledger, which is located on each node, based on a timeframe
- Allow nodes to create blocks and send, receive, or verify transactions

Public Blockchain

Public blockchains are open source. Every transaction done can be seen and verified by anyone. It is considered as fully decentralized where no entity is responsible for or can control
the transactions. A consensus mechanism is in place to secure these blockchains and enforce trust between users. Also, users are given rewards that encourage them to use and mine these blockchains. Bitcoin and Ethereum are the most famous public blockchains (Goranovic et al., 2017). In terms of performance, public blockchains focus more on security and can only perform a few transactions per second (Scherer, 2017).

![Public Blockchain](image)

**Figure 2.2: Public Blockchain**

**Private Blockchain**
Public blockchains are not always suitable for businesses. Therefore, private blockchains were adopted in businesses that want to employ and use this technology. The main difference is that permissions must be granted by the company/enterprise to specific users who can join and use the blockchain, thus performance could be improved compared to the public type (Pongnumkul et al., 2017). Dinh et al., (2017) introduced a tool called ‘BLOCKBENCH’ that was used to evaluate some private blockchains in the market and compared them in terms of data processing workload to traditional relational database systems. It is concluded that
blockchain performance still has a long way to go to match current storage systems for big data processing. Multichain (Greenspan, 2015) and Hyperledger (Blummer et al., 2018) are two famous private blockchains.

![Figure 2.3: Private Blockchain](image)

**Consortium/Federated Blockchain**

The consortium blockchain contains authorized nodes. These nodes are preselected and are the ones responsible for participating in the consensus process, unlike public blockchains where every user is an authorized node. More than one entity or organization manages this blockchain which makes it partially decentralized. The performance is greater than the public type, it has a low transaction cost, high speeds and it is controllable (Zhao et al., 2019). Corda (Brown et al., 2016) and Quorum ("ConsenSys Quorum," n.d.) are two examples of consortium blockchains.
Hybrid Blockchain

Hybrid blockchain is a combination of public and private blockchains. Both characteristics are used where it can be a public or a private permission-based system. The hybrid system allows users to control permissions on the data stored and they can also transfer private transactions to the public blockchains for verification. This method increases the security of the network (“Types of Blockchains,” 2019). An example of such type is Dragonchain (Dragonchain Hybrid Blockchain Solutions and Services, 2017).

2.2 Blockchain Architecture

From its name, a blockchain is a chain of blocks, where lists of transactions are stored. Figure 2.5 displays a continuous sequence of blocks. Each block has only one parent block and the hash of the previous block (e.g., Block $i-1$ in Figure 2.5) is set in the header of the current block (Block $i$) and so on (Zheng et al., 2017).
2.2.1 Block

A *genesis block* is the first block in the blockchain which has no parent. The software version used to generate the block is set in the *block version* zone. Then, the *Merkle tree root hash* contains the hash of all the transaction hashes in the block. Noting that the time it takes to hash one or one-hundred transactions is the same because transactions are hashed indirectly in the Merkle tree. *Timestamp* is the current timestamp of the block as seconds in UTC since 01-01-1970. Next, the *nBits* field contains the target threshold of a valid block hash. In case the SHA256 hash of the block header is greater than this threshold, then the block will not be accepted by the network. This is the mathematical puzzle that needs to be answered by the miners to add the block to the blockchain. The value increases when the difficulty is reduced. The *nonce* starts with ‘0’ and is increased on each hash calculation. Last, in the header, the *Parent block hash* is a 256-bit hash pointing to the previous block. As for the block body, it consists of the transaction counter and the block transactions. The maximum number of transactions in a block is determined depending on the size of the block and each transaction (Kuo Chuen, 2015). Figure 2.6 shows the block structure.
Chapter 2: Background and Motivation

2.2.2 Cryptography

Before discussing the details of consensus algorithms, this section describes the following cryptography concepts: cryptographic hash functions, asymmetric-key cryptography, and digital signature.

2.2.2.1 Cryptographic hash functions

The hash functions technique provides a way for users to verify that an input has not been tampered with. This technique can be applied to a block of data of any size, and it will produce a fixed-length output. This output is always the same for the given input when used with the same hash function. The requirements for such functions are: (Stallings & Brown, 2015)

- The hash of any input x must be relatively easy to compute
- It is a one-way hash meaning that it is computationally infeasible to find x such that Hash(x) = h (where h is the unique output).
- It is weak collision-resistant, meaning that given x an input, it is computationally infeasible to find another input that results in the same output.
• It is strong collision-resistant, meaning that it is computationally infeasible to find two
different inputs producing the same hash output.

Different hash functions exist, but many blockchain implementations use the Secure Hash
Algorithm (SHA) that produces an output size of 256 bits that is presented as a 64-character
hexadecimal string. These functions are used for different tasks in the blockchain such as the
generation of the parent block hash seen in the previous section, the address derivation mostly
used to define the sender and receiver endpoints in a transaction, and different unique
identifiers (Yaga et al., 2018).

2.2.2.2 Asymmetric-key cryptography

Known also as Public Key cryptography that was first suggested by Diffie and Hellman in
1976. Each user will generate two keys, one public and the other private. The data encrypted
by the private key can be decrypted by the public and vice-versa (Diffie & Hellman, 1976).
This method provides trust between peers and enables them to verify the integrity and
authenticity of the transactions while keeping the transaction public. The transaction is said to
be ‘digitally signed’ (Rivest et al., n.d.). In a blockchain, private keys are used to digitally
sign transactions and derive addresses. To verify those signatures, public keys are used. Also,
this technique proves that the private key is owned by the sender. In public blockchains,
everyone can verify the transactions since the public keys are shared with all the users (Yaga
et al., 2018).

2.3 Consensus Algorithms

To reach a consensus in an untrusted network, blockchain exploits the theory behind the
Byzantine Generals Problem. This problem talks about a group of generals commanding an
army circulating a city that needs to communicate to initiate an attack or not. Though, the environment is untrusted as traitors can send fake messages (Lamport et al., n.d.). The same could happen in blockchain networks, therefore setting protocols to prevent no trust between nodes, and to guarantee that records on the distributed nodes are similar is required. Three consensus algorithms will be discussed in the sections that follow. However, more consensus algorithms exist like ‘Delegated Proof of Stake’, ‘Proof of Luck’, ‘Proof of Capacity and Proof of Space’, ‘Proof of Activity’, ‘Proof of Burn’ and ‘Proof of Importance’. Most of these are a variant of the ‘Proof of Work’ and ‘Proof of Stake’.

2.3.1 Proof of Work (PoW)

Bitcoin networks use the proof-of-work algorithm (Nakamoto, 2008). The authentication requires to solve a hard-computational puzzle. The algorithm necessitates that a computed value generated by each node of the network must be less than a given threshold. The hash value of the block header is calculated continuously via distinct nonces by all the users until the target is found, this action is called mining, and the users are called miners. Upon obtaining the right value by a node, all other nodes should approve that it is the correct value. A fraudulent transaction will directly be rejected, else, the collection of the transaction will be added to a new block. An incentive mechanism is used in this system to reward miners since mining is time-consuming (Nakamoto, 2008). In a Bitcoin network, the reward is a small bitcoin currency amount but in different networks, the rewards are different.

A special case might be encountered, where two valid blocks are generated simultaneously by multiple nodes. This will create a fork as seen in Figure 2.7.
Chapter 2: Background and Motivation

Assuming that the fork happened while validating B11 and G11, the miners will keep working on both blocks. But after a while, miners working on G11 and G12 will divert back to B12, because the consensus takes the longer chain as the authentic one. Block G11 and G12 are called orphan blocks. Furthermore, for two competing forks to generate a new block at the same time is very unlikely. The size of blocks needed by the consensus to follow a fork differs between existing blockchains. In Bitcoin, for example, six blocks are needed to be generated. Also, approximately 10 minutes is needed to generate a block, while only 17 seconds are needed in Ethereum (Wang et al., 2018).

2.3.2 Proof of Stake (PoS)

Proof of Stake consensus dictates that users can validate block transactions based on their wealth. This consensus has variant protocols depending on different implementation, to increase decentralization and reduce the impact of the double-spending problem. As an alternative for miners in PoW, PoS has validators that invest a stake in the system to get higher chances in mining the next block. Even so, richer validators can have higher chances, it is not a certainty because validators are chosen randomly by the system (Wahab & Memood, 2018). Any attempt to elude the system, the validator loses his stake. PoS is very efficient and does not need special hardware to be able to do a computational analysis comparing to PoW.
PPcoin is an example of a blockchain using the PoS algorithm (King & Nadal, 2012). The two main types of PoS are chain-based and Byzantine-Fault-Tolerant. The chain-based system enables a random block creator to add a new block within a prefixed time. Then, most blocks are stacked on the top of the blockchain. On the other hand, BFT-based systems randomly select proposers between validators to take part in reaching the consensus. The validators will vote on the proposed blocks multiple times until a block is selected and added to the blockchain (Kostal et al., 2018).

2.3.3 Practical Byzantine Fault Tolerance (PBFT)

This algorithm was introduced before the blockchain era. It can tolerate Byzantine faults in an asynchronous environment in contrast to other algorithms that were designed for synchronous systems. The PBFT algorithm consists of three phases, Pre-prepare, Prepare and Commit, to reach a consensus. Figure 2.8 shows the phases with four replicas (Castro & Liskov, 1999).

![Figure 2.8: PBFT Phases (Castro & Liskov, 1999)](image)

Line C represents the client, line 0 is the primary replica and line 3 is the faulty one. Once the client sends requests to the primary replica, the protocol will start accordingly. In the pre-prepare phase, replica 0 sends a copy of the request for all replicas 1, 2, and 3. In the next
phase, only replicas 1 and 2 reply with a prepare message to all other replicas. Then, the commit message is sent by replicas 0, 1, and 2 without the faulty replica 3. Lastly, the client C will receive a sequence of numbers from replicas 0, 1, and 2. In general, this consensus can handle up to \((n – 1)/3\) faults simultaneously, where \(n\) is the number of replicas (Wu et al., 2019). The PBFT is used in private blockchains, to provide high-throughput transaction while such networks are smaller than public ones. Hyperledger is a private blockchain known to use this algorithm (Lai, 2018). In Table 2.1, a comparison between PBFT, PoS, and PoW is illustrated.

<table>
<thead>
<tr>
<th>Table 2.1 Consensus Algorithm Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Blockchain Type</strong></td>
</tr>
<tr>
<td>Concept</td>
</tr>
<tr>
<td>Transaction Latency</td>
</tr>
<tr>
<td>Mining</td>
</tr>
<tr>
<td>Energy Efficient</td>
</tr>
<tr>
<td>Applications</td>
</tr>
</tbody>
</table>

### 2.4 Smart Contracts

Smart contracts were also proposed long before the blockchain thrive. A smart contract is a special program that runs transactions following some terms of a contract (Szabo, 1997). By using these contracts, the terms of an agreement are performed when the contract conditions are satisfied. It will be executed automatically by the program (Morabito, 2017). Blockchain 2.0 was coined once smart contracts are added into it. A general example of a blockchain smart contract is shown in Figure 2.9.
The contract will be executed by all nodes on the blockchain and consists of a program, storage file, and an account balance. A contract can be created by submitting a transaction on the chain. Published contracts are fixed, and the code cannot be changed. The program of the contract will be invoked on message reception by a user or by other contracts, and its result is validated by the miners to update the blockchain accordingly. Plus, contracts can receive cryptocurrencies and send them to other users or contracts. The storage file is placed on the public blockchain and can be read from or written to while the code is being executed (Delmolino et al., 2016). Two types of smart contracts are identified, deterministic and non-deterministic. Some smart contracts need external data to function properly but blockchains usually limit external data (Morabito, 2017). Deterministic smart contracts rely only on the information found on the blockchain. For instance, users can send money to a lottery contract that will be executed after a period. A random winner will be chosen, and the money is sent to him. In such a contract, all data used are coming from within the blockchain. In contrast, a non-deterministic contract does rely on external data, sometimes referred to as data feeds,
without having the ability to acquire it. Weather contracts could be an example, where based on a defined temperature, the contract will be executed. The temperature is a variable that is not available on the blockchain itself (Alharby & Moorsel, 2017).

Smart contracts are supported by many blockchain technologies available today, each with its limits and implementation. To better understand these differences, three platforms are compared, Bitcoin, NXT, and Ethereum. Bitcoin smart contracts are very limited and complex, they can only support simple contracts (Das et al., 2019). This is due to the scripting language used by bitcoin, for example, loops and limits on withdrawals are not supported, the same code would need to be written multiple times to mimic an iteration which is not efficient (Buterin & others, 2014). Another public blockchain platform, NXT, built-in 2013 and uses PoS consensus, provides smart contracts templates (Hu et al., 2019) that are called ‘smart transactions’. In smart transactions, the scripts are embedded and the code is essentially a software running on the node server (Nxt Developers, n.d.). Fully customized contracts are not supported, and users must use the existing templates. Bitcoin and NXT are not Turing-complete (Lewis, 2016). Following is a more in-depth description of Ethereum smart contracts and its comparison to bitcoin and NXT, since it will be used in Chapter 3:

### 2.4.1 Ethereum

Ethereum is an open-source platform that can be used to develop decentralized applications while implementing smart contracts. Additionally, it has a digital currency called Ether. Ethereum allows the creation of fully customizable contracts that makes it Turing-complete. Currently, Ethereum is one of the top leaders of smart contracts blockchain platforms. This blockchain comes with an Ethereum Virtual Machine (EVM) that is a runtime environment for EVM Bytecode (Chen et al., 2018). The EVM Bytecode is the bottom level of abstraction.
Higher-level languages are used to develop the contracts like Low-level Lisp-like Language (LLL), Serpent, Viper, and Solidity similar to JavaScript and are the most used and adopted (Wohrer & Zdun, 2018). Ethereum uses the Proof of Work consensus, which is the same as bitcoin, but transactions in Ethereum do not always need the same computational power as in bitcoin (Viennas, 2018). To create a smart contract three steps are needed. The first step is to develop a smart contract using the high-level language of choice (e.g.: Solidity). The second step is compiling the code to bytecode via the EVM compiler and in the third step, the compiled code is uploaded to the blockchain using the Ethereum client (Chen et al., 2018). Furthermore, a fee called gas is added to each Ethereum transaction that takes into account the amount of work taken by that transaction to be performed. Gas Limit is a threshold specified by the sender of the transaction providing how much gas is he willing to set. The cost of each unit of gas is called Gas Price. Multiplying the Gas Limit by the Gas Price will generate the total cost of a transaction (Zheng et al., 2020).

**2.5 Model-Driven Engineering**

Most of the software developers tend to build programs by the code-only approach without using models. The abstractions reside in the code and not as a separate entity. This technique could work for small projects but afterward, it will be hard to maintain and understand the business logic and its implementation. Using models reduces system complexity by increasing the level of abstraction, where the model represents an essential feature of the system thus containing less complexity and fewer details (Beydeda et al., 2005). In Figure 2.10 the modeling spectrum is visualized.
Chapter 2: Background and Motivation

19

The model-centric approach is very useful in an unstable development environment where the programming language might change since the business logic is reusable and only the implementation must be adapted. In contrast, the code-centric approach will require a complete rework of the system (Afonso et al., 2006). A software development methodology focused on a model-centric approach is Model-driven engineering (MDE). MDE focuses on domain models that are conceptual models describing a certain problem (Frankel, 2003). To define a ‘model’, three criteria must be met (Ludewig, 2003):

- **Mapping Criterion**: An original object must be related to the model and is referred to as ‘the original’
- **Reduction Criterion**: The model is a reduced form of the original. Yet, some properties of the original must be reflected in the model.
- **Pragmatic Criterion**: The model should be useful and can replace the original in some cases.

![Figure 2.10: Modeling Spectrum (Beydeda et al., 2005)](image-url)
However, this is a very broad definition. In the context of MDE, Kuhne defines the model as ‘From this perspective, a model is an artifact formulated in a modeling language, such as UML, describing a system through the help of various diagram types’ (Kühne, 2006).

In Figure 2.11, the relation between the three model-driven approaches is displayed.

![Figure 2.11: Relationship Between Acronyms of Model-Driven Software (Smith et al., 2015)](Diagram)

MDE increases the level of abstraction by using models and can help in applying concepts closer to the domain of a problem (Bjorn, 2018). Moreover, it is regarded as a software engineering paradigm that does not have concrete support tools. Going into the second approach, Model-Driven Development (MDD) focuses on requirements, analysis and design, and the implementation disciplines. As for Model-Driven Architecture (MDA), it is proposed by Object Management Group (OMG) and focuses on the definition and transformations of the models (Rodrigues da Silva, 2015). MDA is described in the next section and is a key component of this research.
2.5.1 Model-Driven Architecture

MDA was adopted as a new framework by OMG in 2001. Some of OMG’s other technologies are Unified Modelling Language (UML) and XML Metadata Interchange (XMI) (Sacevski & Veseli, 2007). MDA is model-driven as it helps in utilizing models to direct the course of the software development cycle (Karakostas, 2008).

Three default models for a system are identified by MDA, Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM). The relation between these models is represented in Figure 2.12.

![Figure 2.12: MDA Principles (Brambilla et al., 2017)]
The CIM represents the work that a system is expected to do while keeping the implementation aspect hidden by suppressing the technology details and specifications. It also reduces the gap between the domain experts and the information technology used to implement the system. The PIM is a view based on the CIM that describes a system from a platform-independent angle without giving technical details so that it can be mapped to one or more platforms. The combination of the specifications of the PIM is done by the PSM. Enough information must be present to enable logic implementation and code generation. In case the required details are not presented completely by the PSM, it will be considered as abstract (Truyen, 2006).

2.6 Previous Work in the Subject

Using a model-driven approach to smart contract development in blockchain technology is a relatively new subject. A lot of research has been conducted on this subject, but still, no clear architecture is defined due to the diversity of that technology. In this section, previous researches that mostly resembles the work done in this thesis is discussed, specifically the transformations between PIM to PSM to code.

Three approaches to model-driven smart contract development were defined by Boogaard (2018) based on the literature review that they did. These approaches will be reviewed in addition to new resources that are now available, and a fourth new approach that relates to our method is examined.
2.6.1 Agent-Based Approach

Frantz & Nowostawski (2016) proposed a modeling method that automates the translation of institutional constructs into codified machine-readable contractual rules and used a domain-specific language to achieve it. This method is based on the concept discussed in the paper Grammar of Institutions (Crawford & Ostrom, 1995), where institutions are decomposed into rule-based statements to describe the main institution’s function. This Grammar is applied in the context of agent-based modeling (Smajgl et al., 2010). Agent-based models (ABMs) are computational models that contain the interactions and specific representations of the entities of the system modeled (Gilbert, 2019). The statements are built using different components that include: ‘Attributes’, ‘Deontic’, ‘Aim’, ‘Conditions’ and ‘Or else’; ADICO in short. The authors developed a domain-specific language using Scala to transform the ADICO statements into a Solidity code. ADICO statements can be considered here as the PIM. Therefore, a smart contract can be placed using ADICO syntax that is understandable by technical and non-technical people and then derives the Solidity code from it. However, the generated code is only a skeleton contract that does not execute in Solidity and needs a developer to add the actual semantics linked to the actual requirement. Additionally, the work done in this paper does not mention the usage of such approach with different platform other than Solidity. Table 2.4 shows a summary of this approach in addition to the UML approach discussed in Section 2.6.4.

2.6.2 Business Process-Based Approach

This approach is directed to business processes used in the industries. Many papers and tools are created under this approach. Weber et al. (2016) addressed the lack of trust in collaborative process execution in the blockchain and propose an automated way to generate
smart contracts using a translator. This is done by taking the process specification defined in the Business Process Model and Notation (BPMN) and generating smart contracts from it. The BPMN, in this case, is the PIM. This will provide an immutable transaction history. The output generated by this process is a factory contract because the translator is called at design time, and participants assigned to roles might not be known in this case. Workflow patterns are followed in the transformation; therefore, a limitation arises that not all elements of the BPMN can be translated.

Another paper that uses the same approach is “Lorikeet: A Model-Driven Engineering Tool for Blockchain-Based Business Process Execution and Asset Management” (Tran et al., 2018) where the authors presented a model-driven engineering tool named Lorikeet that automatically creates smart contracts from specifications in the BPMN. Here also, the BPMN serves as the PIM. The tool consists of a modeler user interface and three back-end components, the BPMN translator, a Registry generator, and a blockchain trigger. The BPMN translator takes as input a business process model and generates a Solidity smart contract. The trigger also connects with an Ethereum blockchain node to compile, deploy, and interact with the smart contracts. Still, some features in the BPMN model are not supported and for now, the code generation supports Solidity only.

An alternative tool to Lorikeet exists in the industry named Caterpillar. These tools are similar, yet both have their features. The Caterpillar tool is discussed in “Caterpillar: A Blockchain-Based Business Process Management System” (López-Pintado et al., 2017). It is an open-source tool, that supports advanced control flow elements of BPMN like subprocesses, multi-instances, and event handling. However again, code generation is only
supported for Solidity language. A summary and comparison of the BPMN approaches are illustrated in Table 2.2.

<table>
<thead>
<tr>
<th></th>
<th>Weber et al.</th>
<th>Tran et al.</th>
<th>López-Pintado et al</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach</td>
<td>Business processes</td>
<td>Business processes</td>
<td>Business processes</td>
</tr>
<tr>
<td>PIM</td>
<td>Business-Process Based</td>
<td>Business-Process Based</td>
<td>Business-Process Based</td>
</tr>
<tr>
<td>Model transformation</td>
<td>Automated</td>
<td>Automated</td>
<td>Automated</td>
</tr>
<tr>
<td>Result of transformation</td>
<td>Factory Contract</td>
<td>Smart Contract</td>
<td>Smart Contract</td>
</tr>
</tbody>
</table>
| Advantage       | • BPMN is widely used in the industry  
• Automated transformation | • BPMN is widely used in the industry  
• Automatic Ethereum Trigger | • Support advanced BPMN elements  
• Fully Automated |
| Disadvantage    | • Factory contract is not executable  
• Targets only Solidity smart contract | • Complex task  
• Targets only Solidity smart contract | • Complex task  
• Targets only Solidity smart contract |

### 2.6.3 State Machine Approach

The state machine approach is used in different researches to apply the model-driven engineering concepts into smart contract development. In this approach, smart contracts are seen to be acting as state machines. The contract starts with an initial state, and on each transaction done, the state of the contract changes. This method is included in the common patterns of Solidity documentation (*Solidity — Solidity 0.4.24 Documentation*, n.d.). The PIM of this method is the finite state machine (FSM). The strategy followed by Mavridou & Laszka (2018a), was providing a formal model for smart contracts to be modeled, then having clear semantics and finally, a code generator to allow an easy smart contract development with minimal manual coding. A tool was built for that purpose called FSolidM.
The authors also provided another paper to demonstrate this tool (Mavridou & Laszka, 2018b). Though, in that paper, the transformation from FSM to Solidity is semi-automated to guarantee a good code quality; also, some properties are not possible to be modeled in FSM without the use of additional plugins. The details of FSM will not be discussed further in this research. Moreover, the authors built on the above work and created a new framework ‘VeriSolid’ that targets the security aspect of smart contract design (Mavridou et al., 2019). The framework allows developers to verify contracts at a high level, to enable correct-by-design development of such contracts. Still, the framework only generates contracts in Solidity. Before finishing this section, it is worth mentioning a tool that is said to be able to generate smart contracts to different blockchains like Solidity, Vyper, and Yul from a finite state model (Mülder, 2019). However, this tool is still in early development and there are no further details on how it is implemented. Table 2.3 displays a summary and comparison of the FSM approaches discussed.

<table>
<thead>
<tr>
<th></th>
<th>Mavridou &amp; Laszka (a)</th>
<th>Mavridou et al.</th>
<th>Mülder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach</td>
<td>Finite State Machine</td>
<td>Finite State Machine</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>PIM</td>
<td>Finite State Machine Model</td>
<td>Finite State Machine Model</td>
<td>Finite State Machine Model</td>
</tr>
<tr>
<td>Model transformation</td>
<td>Semi-automated</td>
<td>Automated</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>Result of transformation</td>
<td>Smart Contract</td>
<td>Smart Contract</td>
<td>Smart Contract</td>
</tr>
<tr>
<td>Advantage</td>
<td>• Minimal manual coding  • Formal syntax</td>
<td>• Formal syntax • Correct-by-design</td>
<td>• Targets Multiple Blockchain Platforms • Formal syntax</td>
</tr>
<tr>
<td>Disadvantage</td>
<td>• Complex task • Targets only Solidity smart contract</td>
<td>• Complex task • Targets only Solidity smart contract</td>
<td>• Still in early development with no concrete tool</td>
</tr>
</tbody>
</table>
2.6.4 UML Approach

Syahputra & Weigand (2019) used the Unified Modeling Language (UML) and Object Constraint Language (OCL) to define the PIM, and then generate smart contracts on two different blockchains. An existing code generator is used ‘Acceleo’ (Model to Text) (Acceleo Home, n.d.) to produce the smart contract code, after the transformation from PIM to PSM is realized. The authors also followed the commitment-based ontology perspective (de Kruijff & Weigand, 2017) that differentiates between three levels, Essential, Infological, and Datalogical to achieve their goal. Yet, there is a lack of details in this paper where only the models are shown but the platform-specific models and generated code are not discussed. Also, it is not clear if the author’s framework combines all the tools under one package, or the developers will have to use all the different tools used to achieve their aim. The summarized features and comparisons between the UML approach and the Agent Based approach are listed in Table 2.4.

<table>
<thead>
<tr>
<th>Table 2.4 Summary and Comparison – Other Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Approach</td>
</tr>
<tr>
<td>PIM</td>
</tr>
<tr>
<td>Model transformation</td>
</tr>
<tr>
<td>Result of transformation</td>
</tr>
<tr>
<td>Advantage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Disadvantage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
2.7 Research Motivation

There is a lack of research on the use of a model-driven engineering approach to smart contract development. The existing researches related to this subject are focusing on one smart contract language, Solidity, that is currently a top leader in that domain, however, other platforms are catching up and some, do offer solutions to the existing limitations. Additionally, smart contract development is complex, and mistakes can lead to devastating results since most applications are used in the financial domain. Furthermore, the researches discussed in Section 2.6 are complex and the tools used are specific and initially are not designed for blockchain development, such as the finite state machines and the BPMN. Also, developers will have to learn how to use the tools and then work on the smart contract generated by that tool. Again, most of the tools available are focusing on one platform and there are not enough researches on how to use the models on other platforms, keeping in mind that blockchain technologies are evolving and the smart contracts development differs between each platform. This creates a challenge for model transformation from PIM to PSM specifically, that other researches are missing to target.

In this thesis, the focus will be on the transformation from a customized PIM to a completely working source code for two different blockchain networks while keeping the learning curve flat as much as possible for new developers.
Chapter 3: Model-Driven Smart Contracts

This chapter introduces an approach for the development of model-driven smart contracts. A new XML-based programming language and a code generation tool are introduced. The evaluation of the tool is then presented. Finally, the limitations of the language and generator are discussed.

3.1 Introduction

Smart contract development is still in an early phase since the available tools are not yet mature. Even the most famous systems are constantly making changes to their platforms. Due to the diversity of the programming languages used and the difference between each blockchain, developers will have to learn multiple languages and adapt to the changes as fast as possible. However, changing platforms in smart contract development is tricky since each blockchain has its configurations and methods, where sometimes a completely different approach is needed to solve a specific aspect. Tools helping in this transition are very limited and most of them do not target different platforms. Thus, in this chapter, a new XML-based language is presented with its code generator engine that aims to combine the smart contract development into an all in one language, reducing the learning curve for developers and helping them in generating smart contracts for a different platform with a click of a button. The code generation tool is presented alongside its evaluation. The chapter is concluded by discussing some limitations in the current version of the developed tool.
3.2 Smart Contract Development Language

A new language was developed based on the Extensible Markup Language (XML) (Bray et al., 1997). This XML-Based language is used as the PIM (refer to 2.5.1) in this thesis. In the next step, the structure of the XML document had to be defined. The structure is crucial for the code generation phase in Section 3.3 and must support the transformation to different smart contract languages. The class diagram of the XML-based language created is shown in Figure 3.1. The structure, elements, and attributes were inspired by the general smart contract concepts and aspects, and by focusing on Solidity’s documentation (Solidity — Solidity 0.4.24 Documentation, n.d.), since it is the most used and advanced language for smart contract development. Some attributes were also inspired by the Angular framework tags (Angular, n.d.). In Figure 3.1, the classes represent the element tags supported by this new language and their corresponding attributes that can be used with each one of them. The relationships between the classes represent the XML tree nodes. The ‘Contract’ class is the root element that can contain four different child elements that can in turn have children of their own. The ‘Condition’ element supports nested ‘Condition’ elements that will enable the transformation into nested condition statements in the code generator part in Section 3.3. Furthermore, the language has some predefined arguments that can be used as attributes values, which are reserved to the blockchain data manipulation. The elements, attributes, and reserved keywords are discussed next in detail to give a better understanding of what the language offers. The discussion will start from the root element and move downward, passing by each attribute and its accepted values.
**Figure 3.1 XML-Based Language Class Diagram**

**Contract**: contract element is the root element that defines the start of the smart contract section. This element does not support any attribute nor values and must be unique within the XML document. Child elements that are expected to be within a contract are: “ContractName”, “GlobalParameters”, “Modifier” and “Function”.
**ContractName**: the contract name is defined as the text value of that element. Attributes and child elements are not accepted. The “TextValue” illustrated as an attribute in Figure 3.1 represents the element text value.

**GlobalParameters**: All global parameters must be defined under this tag. Only one “GlobalParameters” tag must exist in the contract and should be defined directly after the contract name. Element “Variable” can only be used as a child. However, the “Variable” element attributes can be used to define different types of global parameters.

**Variable (“GlobalParameters” child)**: The attributes that can be used in a “Variable” element under the global parameter node are: “Value”, “AsDefined”, “DType”, “Visibility”, “CustomAssign” and text values. The “Value” is used as the default variable value. The “CustomAssign” should be used when the default value is a combination of statements and not a single value expression. The text value of the element is used as the variable name. The “DType” is the variable data type. The “Visibility” attribute is optional and is set to public by default. The attribute “AsDefined” is used to describe how to use the variable when transformed into the platform-specific code. It has three supported values that are: “Field” as in a regular variable; “Properties” that have a getter and a setter; “Constructor” that can be used to initialize the variable in a constructor. In Section 3.3, some of these properties will be treated the same based on the targeted language and what it supports. The required attributes are: “DType”, “AsDefined” and the element text value.

**Modifier**: This element defines special functions that are mainly used as validations within functions. The modifier should be related to a function. Once a function having a modifier is called, then, the modifier will be triggered, and the modifier condition should be met for the function to work. At present, only authentication modifiers are supported, where the condition
is that the function caller must be the contract owner. This limitation is discussed in Section 3.5. Modifiers have three required attributes: “Name”, “Type” and “Message”.

**Function:** This element represents the start of a function. The attributes supported by a function element are: “Name”, “Argument”, “ArgumentType”, “Payable”, “Visibility”, “Modifier”, “ModifierArg” and “Return”. The “Name” and “Argument” attributes represent the function’s name and arguments respectively. The “ArgumentType” field should contain the data types of the arguments in case these are custom parameters passed to the function. The payable construct must be enabled if the function is expected to receive a money transfer. Again, by default, the “Visibility” is optional and is set to “public” if not specified. The “Modifier” accepts the name of the modifier defined earlier in the contract and is an optional attribute like the “ModifierArg” that accepts the arguments of the modifier’s function. The “Return” construct should contain the return data type of the function. Only “Name” is required, but in case that the “Modifier” attribute exists, then the “ModifierArg” must be added. Function element can have the following child elements: “Variable”, “Line”, “Transfer”, “Return” and “Condition”.

**Variable (“Function” child):** A “Variable” element is a child of a function; it represents a local variable that can be used within the function definition. The “DType” attribute and the text value of this element are similar to the “Variable” element used in the global parameters. This element supports the “DType” and “Assign” attributes only, where “Assign” defines the local variable initial value.

**Line:** Represents a line of code that can either be a regular variable assignment statement or a blockchain event trigger. The behavior of the “Line” element is defined by the attributes available in that element. When a “Field” attribute exists, that means this element will behave
as a regular assigning statement to some variables. The value of the “Field” construct must be a variable defined within the contract. In case the variable is an array, then the element must contain an “Index” to show at which index the assignment must be done and must either have an “Add” or an “Equal” attribute. The usage of “Add” must be followed when a single item is added to an array. In contrast, to subtract or modify the array at that specific index with a custom formula, the “Equal” attribute must be used. For other data types like string, bool, and integers, the “Field” attribute is used with an “Assign” attribute, where “Assign” shows the new value to be assigned in the field selected. The other supported type, “Event”, is used when a blockchain event must be triggered. Events stores the arguments passed in the transaction logs on the blockchains. The available attributes are “Argument” and “Visibility”, where visibility is optional and public by default.

**Return:** The element “Return” must be used when the function has a “Return” attribute defined. It defines the exit statement of the function. It has a “Value” attribute that should contain the variable name returned or the exact value that should be returned.

**Condition:** This element represents a condition statement. It can have two attributes and a text value. The “If” attribute is required and should contain the complete condition statement as a value. This statement can contain the regular operators and the variables defined in the contract or the reserved contracts keywords. The “Catch” attribute means that if the condition is not met then the function will exit. The text value of the element is optional, but it is also used to define a custom message return value in case the condition is met. Condition elements can be nested.

**Transfer:** It is a reserved element that is used when we need to transfer any cryptocurrency. In smart contract development, transferring cryptocurrency must be done using a special
transfer money function. Three attributes are available and required, “From”, “To” and “Amount”.

The reserved variables are: “VAL”, “ADRS”, “OWNER”, “OwnerAuth”, “{{Time.Now}}”, “{{false}}” and “{{true}}”, where “VAL” represents the value of the amount passed in the current transaction; “ADRS” represents the address of the function caller; “OWNER” is the address of the contract creator; “OwnerAuth” is the predefined modifier name currently supported; “{{Time.Now}}” expresses the current time in the blockchain; “{{false}}” and “{{true}}” are the Boolean return types syntax.

A sample smart contract defined in Solidity (Introduction to Smart Contracts — Solidity 0.5.10 Documentation, n.d.) is written in the proposed XML-Based language and shown in Listing 3.1.

Listing 3.1 Smart Contract Sample in XML

1. `<Contract>`
2. `<ContractName>`SimpleStorage`</ContractName>`
3. `<GlobalParameters>`
4. `<Variable AsDefined="Field" DType="uint" Visibility="public">MaxValue</Variable>`
5. `</GlobalParameters>`
6. `<Function Name="setMaxValue" Argument="variable" ArgumentType="uint">`
7. `<Condition If="variable >= MaxValue">`
8. `<Line Field="MaxValue" Assign="variable"></Line>`
9. `</Condition>`
10. `</Function>`
11. `<Function Name="get" Return="uint">`
12. `<ReturnValue="MaxValue"/>`
13. `</Function>`
14. `</Contract>`

The contract displayed in Listing 3.1 describes a simple storage contract where the contract has two functions, one that shows the maximum value stored in an integer variable (get) and the second (setMaxValue) checks if the sent integer is bigger than the existing one and updates the value.
A complete listing of the supported constructs of the language is shown in Table 3.1 and Table 3.2. Table 3.1 shows the accepted elements and their corresponding supported tags, in addition to the child elements that can be used. It also shows if the element accepts text values.

<table>
<thead>
<tr>
<th>Table 3.1 Accepted Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element</strong></td>
</tr>
<tr>
<td>Contract</td>
</tr>
<tr>
<td>ContractName</td>
</tr>
<tr>
<td>GlobalParameters</td>
</tr>
<tr>
<td>Modifier</td>
</tr>
<tr>
<td>Function</td>
</tr>
<tr>
<td>Condition</td>
</tr>
<tr>
<td>Line</td>
</tr>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>Transfer</td>
</tr>
<tr>
<td>Return</td>
</tr>
</tbody>
</table>
Table 3.2 shows the accepted attributes and their supported values. The “String” value used under “Supported Values/Type” column denotes that the attribute value accepts any text.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Supported Values/Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>String</td>
</tr>
<tr>
<td>AsDefined</td>
<td>Constructor – Property – Field</td>
</tr>
<tr>
<td>Dtype</td>
<td>uint – bool – address – mapping=&gt;addressTUint</td>
</tr>
<tr>
<td>Visibility</td>
<td>public – private</td>
</tr>
<tr>
<td>Name</td>
<td>String</td>
</tr>
<tr>
<td>Type</td>
<td>OwnerAuth</td>
</tr>
<tr>
<td>Message</td>
<td>String</td>
</tr>
<tr>
<td>Argument</td>
<td>Strings separated by “;”</td>
</tr>
<tr>
<td>ArgumentType</td>
<td>uint – bool – address – mapping=&gt;addressTUint</td>
</tr>
<tr>
<td>Payable</td>
<td>Y – N</td>
</tr>
<tr>
<td>If</td>
<td>Condition operators – Strings</td>
</tr>
<tr>
<td>Field</td>
<td>String</td>
</tr>
<tr>
<td>Index</td>
<td>String</td>
</tr>
<tr>
<td>Equal</td>
<td>String and operations</td>
</tr>
<tr>
<td>Assign</td>
<td>String</td>
</tr>
<tr>
<td>Event</td>
<td>String</td>
</tr>
<tr>
<td>Return</td>
<td>bool – uint – String</td>
</tr>
<tr>
<td>Modifier</td>
<td>String</td>
</tr>
<tr>
<td>ModifierArg</td>
<td>Strings separated by “;”</td>
</tr>
<tr>
<td>From</td>
<td>address</td>
</tr>
<tr>
<td>To</td>
<td>address</td>
</tr>
<tr>
<td>Amount</td>
<td>uint</td>
</tr>
<tr>
<td>Add</td>
<td>String</td>
</tr>
</tbody>
</table>

### 3.3 Smart Contract Generation Tool

A code generation tool is built to transform the XML-based language into a platform-specific programming language. The tool is developed using the C# programming language through a
Microsoft ASP.NET MVC project (*ASP.NET MVC Pattern*, n.d.). The XML-based language is built as a platform-independent model of the smart contracts and has enough information to be translated into a platform-specific model. That eased the development of the code generation tool.

The activity diagram in Figure 3.2 depicts the main workflow of the code generator.

![Figure 3.2 Complete Code Generation Activity Diagram](image-url)
3.3.1 XML Validation

Initially, the tool expects the XML-based file as an input. It will start by validating the file extension where “.xml” and “.txt” are the accepted ones. Then, the XML format validation process will start. This only validates the regular XML structure to see if the file has close tags for all open tags and that all the syntax is in a valid XML format. The function uses the XDocument class existing in the C# System.Xml library and tries to load the XML file. It is enclosed by a try-catch statement that will generate and log an error in case the file is not supported. Following that, the XML based language validation will start. The system here will validate every item in the file uploaded to check if all constructs and values are supported by the language to be then generated. The diagram in Figure 3.3 portrays a more detailed workflow of the XML-based language validations.

![Diagram](image)

Figure 3.3 XML-based Language Validation
All of the language-accepted constructs are listed in a separate class that is used within the validation to compare items for validity. For example, supported elements, attributes and reserved words are each predefined in a list of string. Also, keypairs like accepted attributes within an element are stored in a Dictionary list, as shown in Listing 3.2.

Listing 3.2 Accepted Arguments in Elements, Dictionary example

1. static Dictionary<string, string[]> ArgumentElementPair= new Dictionary<string, string[]>
2. {
3.     { "Modifier", new string[] { "Type", "Name", "Message" } },
4.     { "Transfer", new string[] { "From", "To", "Amount" } }
5. }

The system loads the XML file into memory and starts going through it line by line by using the XmlReader class. Many options are available to read through an XML file. However, for optimum performance, the XmlReader class was chosen because it provides a fast non-cashed forward-only access to the XML data (dotnet-bot, n.d.). Though, the initial validation does load the file in an XDocument class to check for specific elements where backward access is needed. Additionally, to benefit from these iterations, lists and dictionaries are filled to be used later in the generation phase instead of iterating again over the XML file, especially where backward access is needed in the case of adding events. Events in most smart contract programming languages must be defined at the beginning of the contract, and the function validating the XML is already going through the file, so the events are known before the generation without the need for additional loops.

Other than testing for supported constructs and values, the data type mismatches like returning a Boolean while the function return type is “string” will log an error. Element tree mismatch is not accepted, e.g., having multiple global parameter elements. Duplicate naming is handled by checking duplicate function, and modifier and variable names. Yet, duplicate
local variable names in different functions will not generate an error. Errors encountered will stop the generation immediately.

### 3.3.2 Code Generation

Once validations pass, the code generation process starts for the blockchain platform that the user chose. Noting that the Graphical User Interface (GUI) of the tool, takes as input the XML file and expects the selection of the blockchain platform before running the validation process and the code generation; the tool is discussed later in Section 3.3.2.3.

The code generation logic is illustrated in Figure 3.4. The current version of the tool developed supports two programming languages for different blockchains. Solidity language that targets the Ethereum Virtual Machine (EVM) (*Ethereum for Developers*, n.d.), and C# language targeting the Neo virtual machine (NeoVM) (*Neo Virtual Machine*, n.d.).

![Figure 3.4 Code Generation Workflow](image-url)
The same workflow is used for both platforms, but each has its classes and functions that are called after the validations. At first, the system checks the compiler version selected by the user to generate the correct code version. Next, the initialization phase is started where the file name and extension are set, and an empty file is physically created on the server. Constructors and events are processed by looping through the lists that are filled while validating the file. Listing 3.4 and Listing 3.7 presents the events iteration on different platforms. The remaining constructs are read again from the XML file and transformed into their chosen language. To handle the transformation of the values fetched from the XML, all values are passed to a special function that contains all the logic to fetch matching values and choose the correct result to be returned. Part of these functions is shown in Listing 3.3 and Listing 3.6 for Solidity and Neo respectively. Afterward, the function generation begins. The system can handle nested conditions, by iterating through the condition elements and their descendent child until the end and keeping a count of the number of nested conditions. This count is used to keep track of how many closing brackets should be added after the iterations are complete. The final code translated while the generation is running is filled in a StringBuilder object. At the end of the generation, that object is written to the file generated in the initialization part. Specific language generations are discussed in Sections 3.3.2.1 and 3.3.2.2, and a code generation of the XML document illustrated in Listing 3.1 is displayed with both languages.

3.3.2.1 Solidity – EVM

Solidity is a statically typed language that was designed for Ethereum EVM. It resembles JavaScript, Python, and C++ (Solidity — Solidity 0.7.1 Documentation, n.d.). The language compiler is updated constantly. Each version has its fixes and some versions are reflected in the language itself where syntax might change. Therefore, in the tool built, the Pragma
version must be selected first. Pragma version enables some compiler features to be used while compiling the code, which will help in keeping the code working even if the compiler is updated. SPDX license identifier is another attribute that must be chosen by the user before the translation. It is used to license the smart contract, especially when deploying it to a public network (Layout of a Solidity Source File — Solidity 0.6.8 Documentation, n.d.).

In the Solidity generation process, the dynamic value translation is performed in a method called “GetReservedDataMapping”, where part of that method is shown in Listing 3.3. Object “UtilityClsList” seen in the listing contains the reserved mappings of the languages and other matching tools.

Listing 3.3 GetReservedDataMapping Solidity Method

```solidity
1. private string GetReservedDataMapping(string _value, string _textValue,
2.   List<Tuple<string, string, string, string>> _dtypeMapping = null,
3.   string _catch = "N") {
4.   string[] _array = _value.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
5.   foreach (var IndividualVal in _array) {
6.     //Suppressed Code Exists Before
7.     if (UtilityClsList.EtheriumDataTypeMapping
8.       .Where(c => c.Key.Contains(IndividualVal)).Count() > 0)
9.       _stringBuilder.Append(" " + UtilityClsList.EtheriumDataTypeMapping
10.      .Where(c => c.Key.Equals(IndividualVal))
11.     .Select(c => c.Value).FirstOrDefault());
12.     else if (IndividualVal == "this")
13.       _stringBuilder.Append(" " + _textValue);
14.     else if (UtilityClsList.EtheriumReservedDataMapping
15.       .Where(c => c.Key.Contains(IndividualVal)).Count() > 0)
17.      .Where(c => c.Key.Equals(IndividualVal))
18.     .Select(c => c.Value).FirstOrDefault());
19.     else if (_dtypeMapping != null
20.       && _dtypeMapping.Where(c => c.ElementName.Equals("Variable")
21.       && c.ValueOrFunctionName.Equals(IndividualVal)).Count() > 0)
22.       _stringBuilder.Append(" " + IndividualVal);
23.   } //Suppressed Code Exists After
24. }
```
At line 7 for example, the “UtilityClsList.EtheriumDataTypeMapping” contains a Dictionary mapping the XML language items to the Solidity language-specific datatype. A special condition exists in Solidity used by the keyword “require” instead of an “if” statement and is supported by this tool. This condition will end the function if the statement is true. Another example from the source code of the tool can be seen in Listing 3.4 that depicts the events translations to Solidity code.

**Listing 3.4 Solidity Event Generation**

```csharp
foreach (var item in _events)
{
    StringBuilder eventline = new StringBuilder();
    eventline.Append("event " + item.Name + ")
    string[] _array = item.Argument.Split(new char[] { '; ' },
    StringSplitOptions.RemoveEmptyEntries);
    int _count = 0;
    foreach (var IndividualVal in _array)
    {
        _count++;;
        var _result = _dtypeMapping.Where(c => c.ValueOrFunctionName.Equals(IndividualVal)
            && c.ElementName.Equals("Variable")
            .Select(c => new { c.DataType, c.ValueOrFunctionName })
            .FirstOrDefault();
        if (_result == null)
            _result = _dtypeMapping.Where(c => c.ValueOrFunctionName.Equals(IndividualVal)
            && c.ElementName.Equals("Function")
            .Select(c => new { c.DataType, c.ValueOrFunctionName })
            .FirstOrDefault();
    eventline.Append(_result.DataType);
    if (UtilityClsList.reservedKeys.Contains(IndividualVal))
        eventline.Append("_var" + _count.ToString()
            + (_count == _array.Length ? "":"","));
    else
        eventline.Append("_" + _res.ValueOrFunctionName
            + (_count == _array.Length ? "":"","));
    }
}
```
At line 28, the “sb” object is the StringBuilder that will be written in the file generated when all generations are done. At line 11, the object “_dtypeMapping” is filled while the XML file is validated and contains a list with variable names, data types, and the element tag containing that variable. Other parts of the code follow the same logic as the above listings.

To demonstrate the code generator output in Solidity, the XML-based example displayed in Listing 3.1 is passed to the generator and the resulting Solidity code is shown in Listing 3.5.

### Listing 3.5 Generated Solidity Code

```solidity
1. // SPDX-License-Identifier: MIT
2. pragma solidity ^0.4.16;
3. /**
4. * Automatic Code Generation
5. */
6. contract SimpleStorage{
7.     uint public MaxValue;
8.     function setMaxValue(uint variable)public {
9.         if (variable >= MaxValue) {
10.             MaxValue = variable;
11.         }
12.     }
13.     function get() public view returns (uint) {
14.         return MaxValue;
15.     }
16. }
```

This contract is fully functional and can be tested on the EVM. Due to the example’s simplicity, the XML-based language seems very similar to the code generated, this is due to the way the XML language was designed, where it was inspired by the Solidity language. However, there are more noticeable differences in the more complex examples that are discussed in Section 3.4.
3.3.2.2 C# - NeoVM

Neo blockchain supports C#, VB.NET, F#, Java, Kotlin, Python, GO, and JavaScript (HurifyAdmin, 2019). The code generator currently supports the C# language. Like the Solidity compiler, the Neo compiler is continuously being updated. Thus, the generator expects a predefined compiler version to be selected. After the generation begins, the prefilled lists by the validation process are used to begin the generation. A major difference between C# and Solidity is that the C# Neo smart contract has a main method that is always triggered when a contract is called. An operation is usually sent as an argument to the smart contract, for it to call the appropriate function accordingly. Therefore, an additional structure is sent to the generation method in the case of Neo that contains a list of the function’s definitions. This list is processed first, and the main method is generated. The remaining code structure is similar to Solidity, in the way that the global variables are processed first, then the events and the functions are lastly done. In Neo, the modifiers are translated to conditions that are checked before calling the appropriate function in the main method of the contract. So, in the case that an authentication modifier is added in the function definition in the XML-based language, a condition will be added in the main method that checks if the contract caller is authenticated before calling the function. The main method also generates conditions before calling any function to test that the number of arguments passed on contract calling is the correct number as expected by the functions being called. Another difference is that the current Neo compiler version 2.9.3 does not support constructors, and global parameters need to be initialized as properties. The “AsDefined” attribute shown earlier will be treated similarly in the C# generation.
The function “GetReservedDataMapping” used in Neo is also responsible for all matching needed to get the related C# code. Listing 3.6 shows a part of that function where the value passed is being tested against a list of reserved Neo blockchain arguments. If found, the system checks if the variable contains open and close square brackets, because some predefined variables must be passed as arguments and are always passed before other custom variables. In C#, those arguments are passed in an array with their index number. The condition content started at line 8 will generate the corresponding argument name and index.

Listing 3.6 GetReservedDataMapping Neo C# Method

```csharp
1. string[] attributeVal = _value.Split(new char[] {' '}, StringSplitOptions.RemoveEmptyEntries);
2. foreach (var IndividualVal in attributeVal){
3. //Suppressed Code Exists Before
5. .Where(c => c.Key.Equals(IndividualVal))
6. .Select(c => c.Value)
7. .FirstOrDefault();
8. if (tempVariable.Contains('[') && tempVariable.Contains(']')){
9. var _startIndex = tempVariable.IndexOf('[') + 1;
10. var tempNumber = tempVariable.Substring(_startIndex, tempVariable.IndexOf(']') - _startIndex);
11. int _castedNumber;
12. bool isNumeric = int.TryParse(tempNumber, out _castedNumber);
13. if (isNumeric){
14. if (_castedNumber <= _funcArgName.Count() - 1)
15. _stringBuilder.Append(" "+_funcArgName[_castedNumber]);
16. else
17. _stringBuilder.Append(" "+_funcArgName[_castedNumber - 1]);
18. }
19. }
20. //Suppressed Code Exists After
21. }
```

Again, the events generation function is shown in Listing 3.7. Comparing it to the Listing 3.4 used to generate the Solidity events shows the similarity between the two, but each uses its platform-specific language to be outputted.
Listing 3.7 Neo Event Generation

```csharp
    foreach (var item in _events) {
        StringBuilder eventline = new StringBuilder();
        eventline.Append("\t");
        eventline.Append(GetTopAttribute("DispName", item.Name) + "n"); // C# Annotation
        if (!string.IsNullOrEmpty(item.Visibility))
            eventline.Append("\t\t" + item.Visibility + " ");
        else
            eventline.Append("\t\t "); // public by default
        eventline.Append("\t\tAction<"); // Events are static
        string[] _ArgumentArray = item.Argument.Split(new char[] { ';' }, StringSplitOptions.RemoveEmptyEntries);
        int _count = 0;
        foreach (var IndividualVal in _ArgumentArray){
            _count++;
            var _result = _dtypeMapping.Where(c => c.ValueOrFunctionName.Equals(IndividualVal))
                && c.ElementName.Equals("Variable")
                .Select(c => new { c.DataType, c.ValueOrFunctionName })
                .FirstOrDefault();
            if (_result == null)
                _result = _dtypeMapping.Where(c => c.ValueOrFunctionName.Equals(IndividualVal))
                    && c.ElementName.Equals("Function")
                    .Select(c => new { c.DataType, c.ValueOrFunctionName })
                    .FirstOrDefault();
            eventline.Append(GetTypeMapping(_result.DataType)
                + (_count == _ArgumentArray.Length ? " > "
                + item.Name + " ;" : " "));
        }
        sb.AppendLine(eventline.ToString());
    }
```

Neo events and functions are static. The remaining code generation follows the same logic.

The code generated for Neo blockchain using the Listing 3.1 XML Model, is illustrated in Listing 3.8.

Listing 3.8 Generated C# Neo Code

```csharp
    // Automatic Code Generation
    using Neo.SmartContract.Framework;
    using System.ComponentModel;
    using System.Numerics;
```
using System;

namespace NeoContract1{

    public class SimpleStorage : SmartContract {
        static BigInteger MaxValue {
            get => Storage.Get(Storage.CurrentContext, nameof(MaxValue)).AsBigInteger();
            set => Storage.Put(Storage.CurrentContext, nameof(MaxValue), value);
        }

        public static bool Main(string operation, params object[] args) {
            switch (operation) {
                case "start":
                    Runtime.Log("Contract Started");
                    break;
                case "setMaxValue":
                    if (args.Length != 1) {
                        Runtime.Log("Wrong Args Number");
                        return false;
                    }
                    return SetMaxValue((BigInteger)args[0]);
                    break;
                case "get":
                    if (args.Length != 0) {
                        Runtime.Log("Wrong Args Number");
                        return false;
                    }
                    Runtime.Log(Get().value);
                    break;
                default:
                    Runtime.Log("Operation Does Not Exist");
                    break;
            }
            return true;
        }
    }

    [DisplayName("setmaxvalue")]
    public static bool SetMaxValue(BigInteger variable) {
        if (variable >= MaxValue) {
            MaxValue = variable;
        }
        return true;
    }

    [DisplayName("get")]  
    public static BigInteger Get() {
        return MaxValue;
    }
}
Another feature supported by this tool for C# generation is the naming conventions of the functions, where the first character is capitalized for function and property names. From the code generated, the main method is dynamically created based on the functions listed in the XML model. C# DisplayName attribute is added to each function with the original name written in the XML, without the first character capitalization.

### 3.3.2.3 Tool Interface

A web-based GUI is built to simplify the tool accessibility. The complete code generator can be deployed on any windows-based server and be accessible to the public without any special tools. The user will need to create the XML file using any text editor and then upload the file to the website and download the generated file instantaneously. The UI is illustrated in Figure 3.5 and Figure 3.6.

![Figure 3.5 Home Page UI](image-url)
The home page of the website expects users to upload an XML model. Other information is also present on the website but is not shown in the home page figure. The top header contains a button that redirects the user to the documentation page where the XML-based language supported constructs are listed.

![Image](image.png)

**Figure 3.6 Blockchain Selection UI**

After uploading the file, the user is prompted to select the target blockchain with the required information like the compiler version and license for Ethereum blockchain. Clicking on the “Generate” button will start the validation and generation process. The system will display a message to indicate whether or not the generation was successful. Errors encountered are logged in a file that can be downloaded if an error is generated. In successful scenarios, a button will show up on the screen to download the generated code.
3.4 Evaluation

The evaluation of the tool was performed on multiple aspects. Two real-world case smart contracts were written in the XML format and then generated for both platforms. Besides, the system was tested for efficiency and scalability.

3.4.1 Real-World Case

Two smart contracts are tested with the tool. Both contracts are related to a real-world general case and are taken from the Solidity documentation (Solidity — Solidity 0.4.24 Documentation, n.d.). The first contract shows an implementation of a simple cryptocurrency. The contract issuer can generate coins as he sees fit and regular users can send coins to each other. It is a simplified example where in this case, users do not need to register an account to be able to send coins. They will just need an Ethereum keypair. This contract can be used for example on an online gambling website. The contract owner would be the person who oversees the online casino. Coins will be generated by the owner and sent to the gamblers. The gamblers will be using these coins to play against each other or against the casino where they will be transferring the coins back and forth. The smart contract trigger will be initiated from the gambling system that should be handling all the functionalities of the system.

In Listing 3.9, this smart contract is written using the newly developed XML-based language.

Listing 3.9 Cryptocurrency Implementation

```
1. <Contract>
2.   <ContractName>Coin</ContractName>
3.   <GlobalParameters>
4.     <Variable Value="Ad1HKAATNeFT5buNg5xspbW68f4XV5ssSw" AsDefined="Constructor" DType="address" Visibility="public">minter</Variable>
5.     <Variable AsDefined = "Field" DType = "mapping=>addressTOuint" Visibility="public">balances</Variable>
6.   </GlobalParameters>
```
At line 7, a function definition is set named “mint” where the function body first checks if the address of the function invoker is the contract owner. If so, the total balance of the cryptocurrency is added by the amount sent within the transaction. The other function at line 11 is used to send coins across without authenticating users. However, the sender cannot send coins that he does not have. Hence, the condition at line 12 checks if the amount is within the available balance that the user owns. Lines 13 and 14 are used to decrease the balance of the sender and increase the balance of the receiver. Finally, an event is called to insert this transaction on the blockchain.

The XML document is uploaded to the website to be generated. The first code generated pertains to the Ethereum blockchain. The code is shown in Listing 3.10.

**Listing 3.10 Solidity Cryptocurrency Contract**

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.4.16;
/*Automatic Code Generation*/
contract Coin{
    address public minter;
    mapping(address => uint) public balances;
    event Sent(address _var1,address _receiver,uint _amount);
    constructor(
        address _minter
    ) public {
        minter = _minter;
    }
}
```
Testing Ethereum smart contracts can be done without any prior setup. An open-source IDE tool called Remix is available online that supports Solidity smart contracts development. It supports testing, debugging, and deployment of Solidity smart contracts (Remix - Ethereum IDE & Community, n.d.). Therefore, the code generated is placed inside the Remix IDE for testing and evaluation. The contract was compiled successfully and then deployed on the test net provided by Remix. The contract functions are tested, and all intended actions are found to be working correctly. Afterward, the same XML is generated for the Neo blockchain, and the resulting code is in Listing 3.11.

**Listing 3.11 C# Cryptocurrency Contract**

```csharp
using Neo.SmartContract.Framework;
using System.ComponentModel;
using System.Numerics;
using System;
namespace NeoContract1{
    public class Coin : SmartContract{
        [DisplayName("sent")]
        public static event Action<byte[], byte[], BigInteger> Sent;
        public static readonly byte[] Minter = "Ad1HKAATNmFT5buNgSxspbW68f4XVSSw".ToScriptHash();
```
11. static StorageMap balances = Storage.CurrentContext.CreateMap("_balances");
12. public static bool Main(string operation, params object[] args){
13.     switch (operation){
14.         case "start":
15.             Runtime.Log("Contract Started");
16.             break;
17.         case "mint":
18.             if (args.Length != 3){
19.                 Runtime.Log("Wrong Args Number");
20.                 return false;
21.             }
22.             Runtime.Log(Mint((byte[])args[0], (byte[])args[1], (BigInteger)args[2]));
23.             break;
24.         case "send":
25.             if (args.Length != 3){
26.                 Runtime.Log("Wrong Args Number");
27.                 return false;
28.             }
29.             Runtime.Log(Send((byte[])args[0], (byte[])args[1], (BigInteger)args[2]));
30.             break;
31.         default:
32.             Runtime.Log("Operation Does Not Exist");
33.             break;
34.     }
35.     return true;
36. }
37. [DisplayName("mint")]
38. public static string Mint(byte[] _arg1, byte[] receiver, BigInteger amount){
39.     if (_arg1 != Minter)
40.         return "Not Authorized!";
41.     balances.Put(receiver, amount);
42.     return "true";
43. }
44. [DisplayName("send")]
45. public static string Send(byte[] _arg1, byte[] receiver, BigInteger amount){
46.     if (balances.Get(_arg1).AsBigInteger() < amount){
47.         return "Insufficient balance."
48.     }
49.     balances.Put(_arg1, balances.Get(_arg1).AsBigInteger() - amount);
50.     balances.Put(receiver, amount);
51.     Sent(_arg1, receiver, amount);
52.     return "true";
53. }
Testing needs more work with Neo smart contract than with Solidity since there are no public tools like Remix to be used for it. The Neo C# compiler is downloaded and configured in Visual Studio 2019, then a project of type “NeoContract” is created and the code generated is added to the contract class automatically generated with the project. The code compiled successfully using the Neo compiler. However, testing the contract on a test net was not applicable since Neo Gas was required to be able to test it, and Gas must be purchased. Nonetheless, by compiling the contract using the Neo compiler proves that the generated code is compatible with the Neo platform.

To further evaluate the language and code generator, another more detailed smart contract is tested. This contract describes an Open Auction (Solidity — Solidity 0.4.24 Documentation, n.d.). Open auction is an auction type where everyone can see the bids made, in contrast to a blind auction where bids are not known until the auction ends. The contract developed can be used in any online open auction system. The general idea is that any user can send his bid within the bidding period specified by the contract. Bids are done with Ether money and each bid is bound to the user. If the highest bid is increased, the previous highest bid money will be returned to the corresponding user. Bids will not be accepted after the period ends, and the winner will receive his money by a manual function call done by the contract owner. The manual trigger is needed in this case since contracts do not support calling themselves. Also, it is better when a new highest bid is accepted, that the previous bid is not sent automatically to the previous bidder but instead, let the user call the withdraw function manually. Sending money automatically has security risks associated with it and might be used to call other untrusted contracts especially for Ethereum blockchain. An activity diagram of the open auction contract is shown in Figure 3.7.
A similar process was followed for this contract. The XML file was created and then passed through the generator to get the appropriate generations. The XML content is shown in Listing 3.12, as for the full code generated in both Solidity and Neo C# are displayed in the Open Auction Generated Code Appendix.

**Listing 3.12 Open Auction XML**

1. `<Contract>`
2. `<ContractName>SimpleAuction</ContractName>`
3. `<GlobalParameters>`
<Variable Value="AdHKAATNmFTsbuNgSxspbw68f4X5sSsw" AsDefined = "Constructor" DType = "address" Visibility="public">beneficiary</Variable>

<Variable AsDefined = "Field" DType = "address">highestBidder</Variable>

<Variable AsDefined = "Field" DType = "uint">highestBid</Variable>

<Variable AsDefined = "Field" DType = "bool">ended</Variable>

<Variable AsDefined = "Field" DType = "mapping">addressToUint</Variable>

<Variable AsDefined = "Constructor" DType = "uint">CustomAssign="{{Time.Now}} + this" >auctionEnd</Variable>

<Modifier Name="onlyBy" Type="OwnerAuth" Message="Sender not authorized."></Modifier>

<Function Name="bid" Argument="VAL;ADRS" Payable="Y" Visibility="public">Auction already ended.
  </Function>

<Condition If="{{Time.Now}} >= auctionEnd" Catch="Y">There already is a higher bid.
  </Condition>

<Condition If="highestBid != 0">
  <Line Field="pendingReturns" Index="highestBidder" Add="highestBid"></Line>
</Condition>

<Line Field="highestBidder" Assign="ADRS"></Line>

<Line Field="highestBid" Assign="VAL"></Line>

<Line Event="HighestBidIncreased" Argument="ADRS;VAL" Visibility="public"></Line>

<Function Name="withdraw" Argument="ADRS" Payable="N" Return="bool">amount</Function>

<Condition If="amount >= 0">
  <Line Field="pendingReturns" Index="ADRS" Equal="0"></Line>
</Condition>

<Condition If="!TRANSFER(beneficiary, ADRS, amount)">
  <Line Field="pendingReturns" Index="ADRS" Equal="amount"></Line>
  <Return Value="{{false}}"></Return>
</Condition>

<Function Name="auctionEnd" Argument="" Payable="N" Modifier="onlyBy" ModifierArg="beneficiary">
  <Condition If="{{Time.Now}} < auctionEnd" Catch="Y">Auction not yet ended.
    </Condition>

  <Condition If="ended" Catch="Y">auctionEnd has already been called.</Condition>

  <Line Field="ended" Assign="true"></Line>

  <Line Event="AuctionEnded" Argument="highestBidder;highestBid" Visibility="public"></Line>

  <Transfer From="beneficiary" To="highestBidder" Amount="highestBid"></Transfer>
</Function>

</Contract>
A part of the generated code is shown in Listing 3.13 and Listing 3.14 for Solidity and Neo respectively. The modifier and bidding functions are shown only in the Solidity example.

**Listing 3.13 Open Auction Solidity Partial Contract**

```solidity
modifier onlyBy(address _account){
    require(
        msg.sender == _account,
        "Sender not authorized."
    );
    _;
}

function bid() public payable {
    require(
        now <= auctionEnd,
        "Auction already ended."
    );
    require(
        msg.value > highestBid,
        "There already is a higher bid."
    );
    if (highestBid != 0) {
        pendingReturns[highestBidder] += highestBid;
    }
    highestBidder = msg.sender;
    highestBid = msg.value;
    emit HighestBidIncreased(msg.sender,msg.value);
}
```

The platform-specific matching is converted successfully as seen on line 20, “msg.sender” is a reserved keyword for Solidity and is used correctly. The generated file is then added to the Remix IDE and the contract is compiled and deployed successfully. Tests were done using the IDE also and all auction functions and logic were correct. For Neo, the partial main function and the bid function are shown in Listing 3.14.

**Listing 3.14 Open Auction Neo C# Partial Contract**

```csharp
public static bool Main(string operation, params object[] args)
{
    switch (operation)
    {
```
5.      case "bid":
6.          if (args.Length != 2){
7.              Runtime.Log("Wrong Args Number");
8.              return false;
9.          }
10.         Runtime.Log(Bid((BigInteger)args[0], (byte[])args[1]));
11.         break;
12.      case "auctionEnd":
13.          if (args.Length != 0){
14.              Runtime.Log("Wrong Args Number");
15.              return false;
16.          }
17.          if (Runtime.CheckWitness((byte[])args[0])){
18.              Runtime.Log(_AuctionEnd());
19.          } else{
20.              Runtime.Log("Sender not authorized.");
21.          } break;
22.      }
23.  return true;
24. }
25. [DisplayName("bid")]
26. public static string Bid(BigInteger _arg1, byte[] _arg2)
27. {
28.     if (Runtime.Time >= AuctionEnd)
29.     { 
30.         return "Auction already ended.";
31.     }
32.     if (_arg1 < HighestBid)
33.     { 
34.         return "There already is a higher bid.";
35.     }
36.     if (HighestBid != 0)
37.     { 
38.         pendingReturns.Put(HighestBidder, HighestBid);
39.     }
40.     HighestBidder = _arg2;
41.     HighestBid = _arg1;
42.     HighestBidIncreased(_arg2, _arg1);
43.     return "true"; 
44. }
As discussed earlier in this section, the modifier in Neo is translated to a condition placed inside the main function; line 17 shows the modifier condition. The complete contract is then added to a “NeoContract” project in Visual Studio and compiled. Again, the compilation was successful, and all constructs are converted.

Both cryptocurrency and open auction contracts were converted from the XML-based language to their respective platform-specific language accurately. The next step is evaluating the code generator efficiency and scalability.

### 3.4.2 Efficiency, Scalability, and Line of Code Comparison

Following the confirmation that the code produced is fully functional, the tool is tested for efficiency. A special condition was added to the generator to support bulk tests, in which the duplicated naming validations are stopped. This allowed the duplication of the functions in the same contract to be able to generate contracts with large XML inputs without the need of manually coding the complete XML. All tests were conducted on the same machine with the following specifications:

- **Machine**: Lenovo Ideapad Y510p
- **Operating System**: Windows 8.1
- **RAM**: 8GB DDR3
- **CPU**: Intel Core i7-4700MQ – Base Clock Speed 2.4 GHz
- **GPU**: Nvidia GT750M 2GB GDDR5

The clock speed on this laptop is modifiable, thus, the same tests were performed on two different clock speeds. The first set of test results are shown in Figure 3.8, and the second set is displayed in Figure 3.9. The clock speed is the only difference between the two.
Chapter 3: Model-Driven Smart Contracts

**Figure 3.8** Generation Time vs Model Number of Lines (1.87 GHz)

**Figure 3.9** Generation Time vs Model Number of Lines (3.29 GHz)
The line chart shows the time needed in seconds to validate and transform the XML written contracts to both Solidity and C# code. The same XML was used as an input for each test, and functions were duplicated to achieve bigger inputs on each round. Additionally, cashing was not used, and each test was carried out independently by restarting sessions between each test and the other. A summary of the complete test values is listed in Table 3.3.

<table>
<thead>
<tr>
<th>XML Lines of Code</th>
<th>Neo C# Validation &amp; Generation Time (Seconds)</th>
<th>Solidity Validation &amp; Generation Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clock: 1.87GHz</td>
<td>Clock: 3.29GHz</td>
</tr>
<tr>
<td>40</td>
<td>0.10</td>
<td>0.07</td>
</tr>
<tr>
<td>500</td>
<td>0.11</td>
<td>0.07</td>
</tr>
<tr>
<td>5,000</td>
<td>0.41</td>
<td>0.25</td>
</tr>
<tr>
<td>10,000</td>
<td>1.22</td>
<td>0.68</td>
</tr>
<tr>
<td>15,000</td>
<td>2.36</td>
<td>1.36</td>
</tr>
<tr>
<td>26,000</td>
<td>5.99</td>
<td>3.44</td>
</tr>
<tr>
<td>37,000</td>
<td>11.68</td>
<td>6.63</td>
</tr>
<tr>
<td>54,000</td>
<td>22.99</td>
<td>13.75</td>
</tr>
<tr>
<td>60,000</td>
<td>25.09</td>
<td>16.07</td>
</tr>
<tr>
<td>65,000</td>
<td>33.90</td>
<td>19.20</td>
</tr>
<tr>
<td>75,000</td>
<td>38.35</td>
<td>27.48</td>
</tr>
<tr>
<td>80,000</td>
<td>49.83</td>
<td>28.28</td>
</tr>
<tr>
<td>85,000</td>
<td>61.14</td>
<td>31.60</td>
</tr>
<tr>
<td>90,000</td>
<td>68.05</td>
<td>35.78</td>
</tr>
<tr>
<td>95,000</td>
<td>78.24</td>
<td>38.92</td>
</tr>
<tr>
<td>100,000</td>
<td>101.29</td>
<td>58.97</td>
</tr>
</tbody>
</table>

It is obvious that for small contracts up to 10,000 lines, are generated almost instantly or within seconds at both clock speeds. Smart contracts are not meant to be bloated and are
usually simple enough. Even for complex systems, multiple smart contracts will be generated instead of coding everything in one contract. Nonetheless, the generator proved efficient and scalable even for very large numbers and the output increased at a steady rate while increasing the input. It is also noticeable that from 20,000 lines of code, the Solidity generation is faster than the Neo one. This is due to the output file generated and the final code structure where C# contracts need a lot more lines of code to develop a contract compared to Solidity. Tests are performed on the open auction contract previously discussed. Another important evaluation is the number of lines needed to produce the same contract on the various platforms. Figure 3.10 displays the difference between those numbers.

![Figure 3.10 Number of Lines of Code Needed for the Same Contract on Different Platform](image)

The percentage difference between the XML-based language and Solidity is 60.86%, and 133.05% between XML and C#. These numbers also support the fact that Neo contracts take more time to generate.
The difference between the lines needed to generate the same contract is big and using the XML-based language decreases drastically the lines needed to be written by a developer to achieve the same output. Moreover, the same contract developed through the XML language can be used on different blockchains, hence less development time from different aspects and code reusability is achieved. Also, this approach protects against changes in technology where the code generator can easily be modified to support any new changes without the need of re-developing the whole contract, and by only using the already developed XML contract. In the case of shifting technologies, this approach also thrives where nothing needs to be changed. On another hand, the learning curve for new developers would be much faster since they only need to learn one programming language that supports other platforms without the need of learning two separate languages.

3.5 Threats to Validity and Limitations

Following the various tests conducted in this chapter, the language and code generator developed proved to work under various conditions. Although the language is not complete, it currently supports a significant number of constructs that make it feasible to use for various scenarios. Several compiler versions are currently not supported but can be added at a later stage without having to modify any previously created XML files. However, there might be some minor additions or changes to the language as it becomes more mature and supports other smart contract languages. Custom modifiers are not supported but can be replaced with a simple condition that can achieve the same output.

The advantages of such a system at its current state outweigh the limitations. The maturity of the system will produce a much stable and performant outcome.
Chapter 4: Conclusion

This chapter summarizes the main results, contributions, and possible extensions to the work that can be done in the future.

4.1 Summary of the Main Results

A new XML-based language was built with its code generator that supports the development of smart contracts and their translation to two different smart contract languages, Solidity and C#. Two real-world smart contracts were used to evaluate the language. Both contracts were successfully transformed into their corresponding target platform. The Solidity code was tested in Remix IDE and proved to be working. The C# code was compiled successfully using the Neo compiler. Then, efficiency and scalability tests were conducted. The code generation proved to be efficient in different scenarios and contracts. For example, contracts with less than 10,000 lines of code are generated in under a second. The generator was able to handle also very large files in an acceptable amount of time by using a regular laptop for testing. No caching was used which indicates that the system is not depending on any caching method to become efficient.

4.2 Main Contributions of the Thesis

Using the tools proposed in this thesis grant many advantages. Developers can build smart contracts with less time than essentially needed by having to code much fewer lines. New developers can learn one language and then deploy contracts to different blockchains without
the need to learn two different smart contract languages, thus reducing the learning curve and increasing productivity. Shifting contracts from technology to another can be easily done by using the proposed approach. The language is used as a platform-independent model that hides the language complexity yet can be transformed into the targeted platform by the code generator.

### 4.3 Possible Extensions and Future Work

The system can be extended to support other smart contract languages. Additional constructs will need to be added to support more platform-specific languages and tools. These constructs include inheritance, external contract calls, and additional data types. At a later stage, a pretty print algorithm will be implemented so that users will not need to use third-party tools to beautify the code. Other future work direction would be enabling the system to generate the XML model from platform-specific code.
Bibliography

https://doi.org/10.1109/ACCESS.2019.2902501


https://doi.org/10.1109/MBD-MOMPES.2006.13

https://doi.org/10.5121/csit.2017.71011


Introduction to Smart Contracts—Solidity 0.5.10 documentation. (n.d.). Retrieved September 13, 2020, from https://solidity.readthedocs.io/en/v0.5.10/introduction-to-smart-contracts.html


https://doi.org/10.1109/ICCCN.2017.8038517


https://doi.org/10.1016/j.cl.2015.06.001


Solidity—Solidity 0.4.24 documentation. (n.d.). Retrieved May 2, 2020, from https://solidity.readthedocs.io/en/v0.4.24/
Solidity—Solidity 0.7.1 documentation. (n.d.). Retrieved September 12, 2020, from https://solidity.readthedocs.io/en/v0.7.1/


Types of Blockchains – Decide which one is better for your Investment Needs. (2019). In Types of Blockchains—Decide which one is better for your Investment Needs—DataFlair. https://data-flair.training/blogs/types-of-blockchain/


Appendix A: Open Auction Generated Code

The following listing shows the complete open auction code generated for Solidity.

Listing A.1 Open Auction Solidity Code Generated

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.4.16;
/** *
* Automatic Code Generation
*/
contract SimpleAuction{
    address public beneficiary;
    uint public auctionEnd;
    address public highestBidder;
    uint public highestBid;
    bool public ended;
    mapping(address => uint) public pendingReturns;
    event HighestBidIncreased(address _var1,uint _var2);
    event AuctionEnded(address _highestBidder,uint _highestBid);
    constructor(
        address _beneficiary,
        uint _auctionEnd
    ) public {
        beneficiary = _beneficiary;
        auctionEnd = now + _auctionEnd;
    }
    modifier onlyBy(address _account) {
        require(
            msg.sender == _account,
            "Sender not authorized."
        );
    }
    function bid() public payable {
        require(
```
now <= auctionEnd,
   "Auction already ended."
);
require(
    msg.value > highestBid,
   "There already is a higher bid."
);
if ( highestBid != 0) {
    pendingReturns[highestBidder] += highestBid;
}
highestBidder = msg.sender;
highestBid = msg.value;
emit HighestBidIncreased(msg.sender,msg.value);
}

function withdraw() public returns (bool) {
    uint amount = pendingReturns[msg.sender];
    if ( amount >= 0) {
        pendingReturns[msg.sender] = 0;
        if (!msg.sender.send(amount)) {
            pendingReturns[msg.sender] = amount;
            return false;
        }
    }
    return true;
}

function auctionEnd() public onlyBy( beneficiary) {
    require(
        now >= auctionEnd,
   "Auction not yet ended."
    );
    require(
        !ended,
   "auctionEnd has already been called."
    );
    ended = true;
    emit AuctionEnded(highestBidder,highestBid);
    beneficiary.transfer(highestBid);
}
The following listing displays the C# code generated for the open auction smart contract.

**Listing A.2 Open Auction C# Code Generated**

```csharp
// Automatic Code Generation
using Neo.SmartContract.Framework;
using System.ComponentModel;
using System.Numerics;
using System;
namespace NeoContract1
{
    public class SimpleAuction : SmartContract{
        [DisplayName("transfer")]
        public static event Action<byte[], byte[], BigInteger> Transferred;
        [DisplayName("highestbidincreased")]
        public static event Action<byte[], BigInteger> HighestBidIncreased;
        [DisplayName("auctionended")]
        public static event Action<byte[], BigInteger> AuctionEnded;
        public static readonly byte[] Beneficiary =
            "Ad1HKAATNmFT5buNgSxspbW68f4XV5ssSw".ToScriptHash();
        static BigInteger AuctionEnd
        {
            get => Storage.Get(Storage.CurrentContext, nameof(AuctionEnd));
            set => Storage.Put(Storage.CurrentContext, nameof(AuctionEnd),
                value);
        }
        static byte[] HighestBidder
        {
            get => Storage.Get(Storage.CurrentContext, nameof(HighestBidder));
            set => Storage.Put(Storage.CurrentContext, nameof(HighestBidder),
                value);
        }
        static BigInteger HighestBid
        {
            get => Storage.Get(Storage.CurrentContext, nameof(HighestBid)).
                AsBigInteger();
            set => Storage.Put(Storage.CurrentContext, nameof(HighestBid),
                value);
        }
    }
}
```
static BigInteger Ended
{
    get => Storage.Get(Storage.CurrentContext, nameof(Ended)).AsBig
    Integer();
    set => Storage.Put(Storage.CurrentContext, nameof(Ended), value
    );
}

static StorageMap pendingReturns = Storage.CurrentContext.CreateMap
("_pendingReturns");

public static bool Main(string operation, params object[] args)
{
    switch (operation){
        case "start":
            if (Ended != 1 && Runtime.CheckWitness((byte[])args[0]
            )){
                Ended = 0;
                AuctionEnd = Runtime.Time + (BigInteger)args[1];
                Runtime.Log("Contract Started");
            }
            else
            {
                Runtime.Log("Sender not authorized.");
            }
            break;
        case "bid":
            if (args.Length != 2){
                Runtime.Log("Wrong Args Number");
                return false;
            }
            Runtime.Log(Bid((BigInteger)args[0],(byte[])args[1]));
            break;
        case "withdraw":
            if (args.Length != 1){
                Runtime.Log("Wrong Args Number");
                return false;
            }
            return Withdraw((byte[])args[0]);
            break;
        case "auctionEnd":
            if (args.Length != 0){
                Runtime.Log("Wrong Args Number");
                return false;
            }
    }
    if (Runtime.CheckWitness((byte[])args[0])){
        Runtime.Log(_AuctionEnd());
    }
    else{
        Runtime.Log("Sender not authorized.");
    }
    break;
    default:
        Runtime.Log("Operation Does Not Exist");
        break;
    }
    return true;
}

[DisplayName("bid")]
public static string Bid(BigInteger _arg1, byte[] _arg2)
{
    if ( Runtime.Time >= AuctionEnd) {
        return "Auction already ended."
    }
    if ( _arg1 < HighestBid) {
        return "There already is a higher bid."
    }
    if ( HighestBid != 0) {
        pendingReturns.Put(HighestBidder, HighestBid);
    }
    HighestBidder = _arg2;
    HighestBid = _arg1;
    HighestBidIncreased(_arg2,_arg1);
    return "true";
}

[DisplayName("withdraw")]
public static bool Withdraw(byte[] _arg1)
{
    BigInteger amount = pendingReturns.Get(_arg1)
        .AsBigInteger();
    if ( amount >= 0) {
        pendingReturns.Put(_arg1, 0);
        if (!Transfer(Beneficiary, _arg1, amount)) {
            pendingReturns.Put(_arg1, amount);
            return false;
        }
    }
    return true;
[DisplayName("_auctionend")]

class _AuctionEnd()
{
    public static string _AuctionEnd()
    {
        if (Runtime.Time <= AuctionEnd)
        {
            return "Auction not yet ended.";
        }
        if (Ended == 1)
        {
            return "auctionEnd has already been called.";
        }
        Ended = 1;
        AuctionEnded(HighestBidder, HighestBid);
        Transfer(Beneficiary, HighestBidder, HighestBid);
        return "true";
    }

    private static bool Transfer(byte[] from, byte[] to, BigInteger amount)
    {
        //Check parameters
        if (from.Length != 20 || to.Length != 20)
        throw new InvalidOperationException("The parameters from and to SHOULD be 20-byte addresses.");
        if (amount <= 0)
        throw new InvalidOperationException("The parameter amount MUST be greater than 0.");
        if (!IsPayable(to))
        return false;
        if (!Runtime.CheckWitness(from))
        return false;

        StorageMap asset = Storage.CurrentContext.CreateMap(nameof(asset));
        var fromAmount = asset.Get(from).AsBigInteger();
        if (fromAmount < amount)
        return false;
        if (from == to)
        return true;

        //Reduce payer balances
        if (fromAmount == amount)
        asset.Delete(from);
        else
        asset.Put(from, fromAmount - amount);

        //Increase the payee balance
178.     var toAmount = asset.Get(to).AsBigInteger();
179.     asset.Put(to, toAmount + amount);
180.
181.     Transferred(from, to, amount);
182.     return true;
183. }
184.
185. private static bool IsPayable(byte[] to)
186. {
187.     var c = Blockchain.GetContract(to);
188.     return c == null || c.IsPayable;
189. }
190. }
191. }