FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Generating Ethereum Smart Contracts from DasContract Language
Student:	Bc. Jan Frait
Supervisor:	Ing. Marek Skotnica
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

Blockchain smart contracts (SC) are an emerging technology that aspires to change the way people conduct contracts. However, the language of smart contracts is a domain-specific programming language Solidity that is hard to understand by humans and is prone to errors. Based on preliminary research, DasContract models seem to provide a better way to define smart contracts. A goal of this thesis is to propose a way how to generate Ethereum smart contracts from DasContract models.

Steps to take:
1. Explore the state-of-the-art Ethereum blockchain technology and assess its strengths and weaknesses.
2. Analyze ways to generate Ethereum smart contracts from DasContract models.
3. In .NET Core implement and test an algorithm that generates Ethereum smart contracts from DasContract models.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 5, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Generating Ethereum Smart Contracts from DasContract Language

Bc. Jan Frait

Department of Software Engineering
Supervisor: Ing. Marek Skotnica

July 30, 2020

Acknowledgements

Rád bych především poděkoval vedoucímu této práce, Ing. Markovi Skotnicovi, za příkladné vedení, cenné rady, konzultace a trpělivost. Dále děkuji své rodině za podporu, které se mi dostávalo po celou dobu mého studia.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on July 30, 2020 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Frait, Jan. *Generating Ethereum Smart Contracts from DasContract Language*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

V současné době jsou smlouvy mezi lidmi nebo jinými subjekty složitě vymahatelné před soudem a nebo neúměrně komplikované, složité na pochopení a s různými poplatky v průběhu.

Tato práce diskutuje možné použití vizuálního jazyka DasContract spolu s blockchain technologií k vytvoření smluv mezi dvěmi nebo více stranami, které jsou jednoduché na implementaci, decentralizované a bezpečné. V praktické části je pak navržen a implementován generátor mezi DasContract a Solidity jazyky.

Klíčová slova DasContract, smart kontrakt, blockchain, hypotéka, Ethereum, Solidity, smlouva

Abstract

Currently, the contracts between people or other subjects are either hardly enforced in front of court, or disproportionately complicated, hard to understand and with various expenses along the way.

This thesis discusses the possible usage of DasContract visual language combined with blockchain technology to create easy to implement, decentralized and secure contracts between two or more parties. In the practical part, DasContract to Solidity generator is being designed and implemented.

Keywords DasContract, smart contract, blockchain, mortgage, Ethereum, Solidity, contract

Contents

List of Figures

List of Tables

Introduction

Contracts between people or other subjects may take on different forms. From the most simple one – the verbal contract – to the currently most developed one legally binding contract represented by legal text. All of those approaches have one in common – they are not ideal.

Verbal contracts, while legally binding in most of countries, have some serious limitations to them, described in the law in specific country or state. There is also no written record of the contract, which makes it usually very hard to enforce in front of court.

Legally binding contracts are surely better and safer option, although even they have some flaws. There are expenses for lawyer drafting the contract, in some cases even a notary is needed to certify the contract. This process is not only expensive, but also time consuming and lot of people will try to avoid this kind of contract if not necessary. Other problems with legal texts include possible errors made by lawyer drafting the text and legal framework that contains ambiguities[5].

The thesis proposes using different approach with the goal of decentralized ambiguity control of contracts, making them easier to understand and cheaper to draft. Drafting part can be achieved using DasContract visual language specifically designed for this use case, while the ambiguity would be controlled using smart contracts on decentralized blockchain network like Ethereum.

Structure of the thesis

Structure of this thesis is as follows:

- In **Chapter 1** we will take a look at history and current state of blockchain technology and dive deeper into the Ethereum blockchain network, talk about its security, future and and consensus algorithm. We will also explain what is DasContract and what parts it consists of.

- **Chapter 2** compares different generations of blockchain, and advantages and disadvantages of choosing the Ethereum blockchain-
- **Chapter 3** is about analysis of program from practical part of thesis, how it works and how it generates Solidity code from DasContract.
- **Chapter 4** discusses the Solidity code generating flow and implementation of Proof-of-Concept example of mortgage contract.
- The objective of **practical part** of the thesis is to develop program in .NET Core that will convert any DasContract file to corresponding Solidity smart contract code.

Chapter 1

Theoretical background

This chapter discusses the theoretical background of blockchain, Ethereum and DasContract. We will take a look at history of blockchain, its first application as Bitcoin and how it works from cryptographic perspective. How Ethereum changed the view on usage of blockchain, what other projects are trying to surpass Ethereum and why they might (not) succeed. Finally we will discuss DasContract – the model used for generating smart contracts – and its structure.

1.1 Blockchain

The idea behind Blockchain was first introduced by Satoshi Nakamoto in Bitcoin white paper in 2008. Satoshi's identity is still unknown – he communicated with the community around Bitcoin only through forums and other written online forms and he could be either one individual person or a group of people working on the blockchain idea.

The key features of blockchain – security, transparency, decentralization, immutability, and programmability, are combined in a platform, which doesn't need any central authority in order to process transactions, value, and information transfers.

1.1.1 Bitcoin

The best definition of Bitcoin and its use case is by the author Satoshi Nakamoto himself as he wrote in Bitcoin whitepaper:

What is needed is an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party. Transactions that are computationally impractical to reverse would protect sellers from fraud, and routine escrow mechanisms could easily be implemented to protect buy-

ers.[9]

First and the most publicly recognized implementation of blockchain technology is Bitcoin. Bitcoin's main purpose is processing payments in secure way with no need for trusted third party or any kind of escrow service. It is not only first but also one of the simplest implementation of blockchain – even though Bitcoin does technically support smart contracts, they aren't as widely used as e.g. Ethereum's smart contrats. That is because of usage of very primitive OP codes and also high transaction costs.

Bitcoin is a decentralized peer-to-peer network. There isn't any central entity responsible for managing the system. It is a truly democratic platform, open to everyone to participate and contribute. As Satoshi Nakamoto suggests, the main idea is to have a system without intermediaries that prevents double-spending with the sole involvement of peers on the network.[10]

1.1.2 Proof-of-Work

Proof-of-Work algorithm is best described from the Bitcoin whitepaper:

A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based Proof-of-Work, forming a record that cannot be changed without redoing the Proof-of-Work.[9]

Proof-of-Work (PoW) is the original consensus algorithm in a blockchain network used by Bitcoin. Consensus algorithm is how the blockchain network communicates between nodes which transactions to put into new blocks write them irreversibly into the blockchain. With PoW, miners compete against each other to complete mathematical puzzles and first of them with correct solution is awarded by fraction of Bitcoin.

The main working principles are a complicated mathematical puzzle and a possibility to easily prove the solution. These puzzles varies between blockchains and can include, but are not limited to, hash function or integer factorization. As the network is growing, it is facing more and more difficulties.[1] The algorithms difficulty is linearly dependent on number of nodes in blockchain.

How complex a puzzle is depends on the number of users, the current power and the network load. The hash of each block contains the hash of the previous block, which increases security and prevents any block violation as can be seen in Figure 1.1

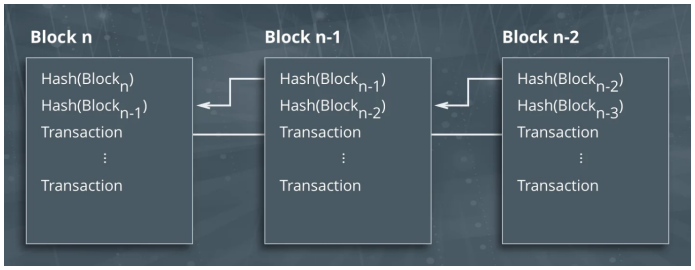

Figure 1.1: Blockchain blocks visualization[1]

1.1.3 Cryptographic techniques used in blockchain

Once again, we will use citation from Bitcoin whitepaper to define electronic coin and cryptography behind them:

"We define an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership."[9]

There are several cryptographic techniques used in blockchain to secure decentralization and privacy. In most of blockchains this includes public and private keys, hash functions and digital signatures.

The private key is used by the sender to sign the transaction, which proves the ownership of the private key without having to reveal it. The public address, which acts like an account, can send or receive funds as well as transact smart contracts. Both funds sending and smart contracts calls have to be signed by user's private key to confirm intention of the transaction. Public address is derived from the public key and can be shared openly without any security exposure. The private key must be stored securely, ideally in offline storage, e.g. hand written on paper in vault, and never disclosed with anyone who isn't the owner of the account since it can't be changed. Only the private key gives access to the user's funds and ability to sign transactions on the blockchain in the name of this account. If the private key is lost, access to the funds is lost permanently as it can't be recovered.

What are blocks and how they are stored in blockchain describes following citation:

Transactions in blockchain ledger are grouped into blocks. These blocks can be different in size, ranging from single transaction to hundreds of them in single block. Transactions themselves also typically vary in the size of the information they contain, smart contract transactions are usually bigger than simple currency transfer. Hence, a blockchain can benefit from some standard-

ization and rationalization of the data it stores. A mechanism that allows us to address that are cryptographic hash functions, which are an efficient way to secure data integrity and reduce file size. Hash functions are used to convert input data of any length into a compressed unique fixed length string of characters (also known as a bit string). This output data serves as a unique reference code or digital fingerprint to verify the authenticity of some underlying dataset without the need to actually check the entire dataset.[10]

1.1.4 Peer-to-Peer Network

The Bitcoin Peer-to-Peer (P2P) network consists of thousands of nodes all over the world and throughout all continents as can be seen on Figure 1.2. As of July 2020 there are 10428 full nodes[2], most of them are running in Europe and USA, although even still developing countries like Cambodia or Suriname make appearance.

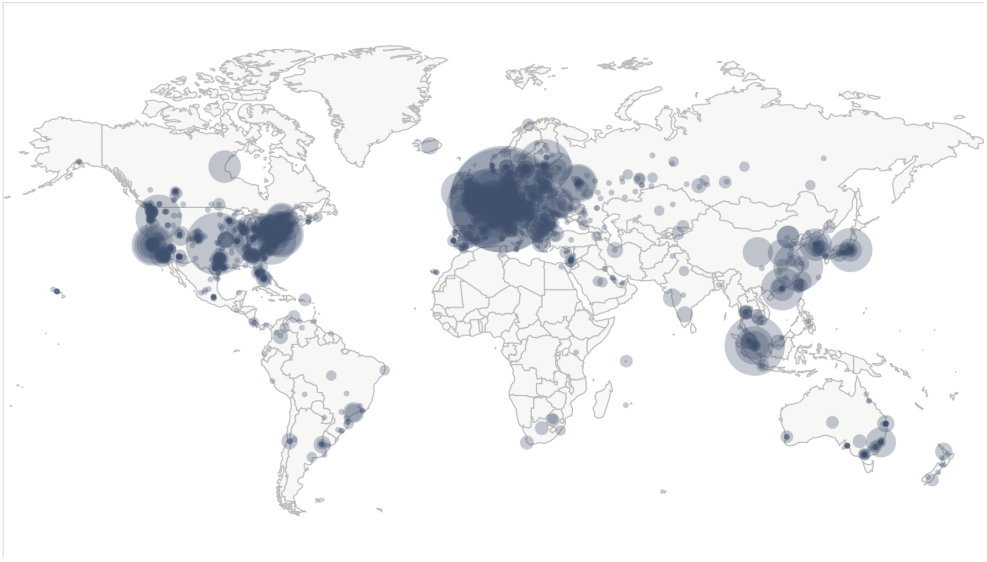

Figure 1.2: Bitcoin nodes distribution[2]

In Bitcoin blockchain, several different types of nodes exist: full nodes, mining nodes, masternodes and lightweight (SPV) nodes. Mining node and masternode are subtypes of full node.

Full nodes are copies of the whole blockchain ledger and can verify transactions without any need of external reference. Since full nodes have every transaction stored, they have information of every asset of every blockchain user and can track history of every coin from the moment of its creation. Every block consists of several transactions and info of its successor. This way the chain of blocks is created and full nodes can go up the chain all the way

from genesis block (the first block in the blockchain) to get the history of every account in the blockchain. On the downside, full nodes are inherently very demanding on storage and bandwidth. Currently, full Bitcoin blockchain has almost 300GB and is rapidly growing in size.

Mining node is subtype of full node. On top of full node's responsibilities, it is also trying to "mine" new blocks. This means solving mathematical puzzles as mentioned in section 1.1.3. The solution is then sent to other full nodes which check correctness and if at least 51% of nodes deem the solution correct, miner can push the block into ledger and receive reward.

The size and bandwidth requirements of full nodes can sometimes be a problem, usually for people using it only as a wallet. This is why lightweight solution exists. Lightweight nodes (also known as SPV – Simple Payment Verification – nodes) don't store the copy of the blockchain. They instead rely on full nodes to provide them all necessary information. Since they don't have full ledger stored locally, they can't verify transactions like full nodes can, going all the way to genesis block. They instead find the block with transaction they need to verify and then check if there are more blocks on top of this one. If there are 6 or more blocks on top, it is generally accepted as verified and irreversible transaction. This way, lightweight nodes are many times less demanding on hardware, however they are relying on full nodes to provide them with all information and therefore sacrificing some security.

1.1.5 Other blockchains and their use cases

Up until now we were talking about Bitcoin's implementation of blockchain. And although the main idea stays more or less the same, the speed of transactions, security, privacy and consensus algorithms may vary between blockchains. We will be talking more about Ethereum in section 1.2, but there are some other blockchain examples and differences between them.

1.1.5.1 Cardano

Cardano's development started back in 2015 and is still yet to be finished to the fully functioning state, mainly because The Cardano Foundation demands multiple code reviews and has no tolerance for any kinds of security bugs after release. The essential elements of blockchain Cardano aims to improve are scalability, interoperability and sustainability. Disadvantage of this approach is huge competition Cardano has in all other blockchains, users and companies might opt to using not optimal solution now rather than optimal one in several years. Cardano is also very popular in academic community.

Cardano uses its own Proof-of-Stake (PoS) consensus algorithm called Ouroboros.[11] Instead of mining blocks like in Bitcoin blockchain, holders of Cardano currency (ADA) can stake their tokens either directly by running node, or indirectly by staking their tokens to stake pool. PoS algorithm

then chooses node based on number of staked ADA tokens who produces the next block and is rewarded by ADA tokens. This makes Cardano much more environment-friendly than Bitcoin's PoW algorithm while still assures decent level of decentralization.

1.1.5.2 EOSIO

EOSIO, usually referred simply as EOS, is blockchain developed by Block.one company, however the EOS blockchain recognized as "mainnet" is run completely by community and Block.one has no authority over it other than voting power of their tokens. With this background, EOS is very controversial project in blockchain community. Concerns of opponents of EOS include:

- Funding of project via ICO (Initial Coin Offering) and EOS token distribution.
- Use of Delegated Proof-of-Stake (DPoS) consensus algorithm with only 21 nodes that have all power over network where most of them are (as of this writing) located in China.[12]
- Written constitution of EOS blockchain that is not implemented in the blockchain itself. That means, it should be honored, however the final decision and action taking is up to those 21 nodes which can (on purpose or not) break it.

Nevertheless if we look away from these issues, EOS is very powerful blockchain with no transaction costs. Instead users "buy" the blockchain's CPU time, RAM memory and storage by staking EOS tokens. Every user can also vote for up to 30 nodes to represent them in decisions same way as people in house of representatives in country's government. However only 21 of those nodes are producing blocks as mentioned above, therefore the level of decentralization is much lower than in Bitcoin network.

1.1.5.3 VeChain

VeChain is the least known blockchain from these three examples, however is currently one of the very few ones that are used commercially. There are already a lot of companies using or testing VeChain to track food or other goods. Solution co-developed by risk management and quality assurance company DNV GL called My Story™is probably the most developed one and is actually in use by some of DNV GL's clients. Companies like BMW, PwC or Walmart[13] ale also testing VeChain as blockchain for tracking their products and providing customers with its history, including origin of the products, tracking history or temperature during transportation.

From technical perspective, VeChain uses dual-token system. The main token determining the market capitalization of VeChain blockchain is VET

which is not mineable meaning you can't get them any other way than purchasing with money. By holding VET in your VeChain wallet, you automatically generate the secondary token called VTHO. This token is used primarily to pay for the transactions on the blockchain, including currency transfers and smart contract fees.

VeChain blockchain uses Proof-of-Authority (PoA) consensus for validating transactions. There are 101 authority nodes which secure stability and to some extent decentralization of network. Authority nodes are selected by VeChain Foundation and need to prove their identity, own at least 25 million VET tokens and actively use or help to develop the blockchain. This means although VeChain isn't as decentralized as Bitcoin or Ethereum, it is much faster to verify transactions and arguably better for supply chain use case.

1.2 Ethereum

Ethereum is a blockchain and decentralized computing platform that allows the execution of smart contracts.

Unlike Bitcoin, Ethereum's purpose is not primarily to be a digital currency payment network. While the digital currency ether is both integral to and necessary for the operation of Ethereum, ether is intended as a utility currency to pay for use of the Ethereum platform as the world computer. The Ethereum platform enables developers to build decentralized applications with built-in economic functions. While providing high availability, auditability, transparency, and neutrality, it also reduces or eliminates censorship and reduces certain counterparty risks.[14]

1.2.1 Ether and ERC tokens

Ethereum blockchain has in its core only one token called ether (ETH), for small fractions of ETH is however used term *gwei* - 1 gwei equals to 0.000000001 of ether. Transaction require *gas* – which refers to the amount of gwei needed – to successfully complete given transaction. Transaction costs differ based on blockchain load and also type of transaction. Sending ETH form one address to another is usually very cheap, on the other hand deploying smart contract to blockchain can be several times more expensive.

There are also other tokens than ETH on Ethereum blockchain. These are called ERC (Ethereum Request for Comment) tokens and while some implementations of them are popular almost as much as Ethereum itself, they aren't native for Ethereum blockchain. ERC tokens are just smart contracts and standards for those tokens were agreed on by the community. The need to achieve consensus on basic blockchain function as having a custom token is considered as one of the disadvantages of Ethereum.

Currently most used standard of ERC token is **ERC20**. ERC20 tokens are fungible – each token is the same and don't have any specific characteristic

– and they are also divisible up to 18 decimals. These two features make ERC20 tokens ideal for payments. ERC20 smart contracts consists of 6 basic functions[15]:

- **totalSupply()** returns number of tokens in circulation.
- **transfer()** is used to initial distribution of tokens from its smart contract address to the destination address.
- **transferFrom()** enables holders to send token to another address.
- **balanceOf()** returns the number of tokens in specific user's wallet.
- **approve()** function ensures that nobody can create additional tokens.
- **allowance()** checks whether or not has sender as many tokens as he wants to send when using transferFrom() function.

The second most used standard is **ERC721** and those tokens are non-fungible and non-divisible. While ERC20 is ideal for payments, ERC721 is designed to represent unique individual objects ranging from virtual pets as shown by CryptoKitties project all the way to real houses if implemented by authorities responsible for real estate evidence. Each token may then have very different value than another one.

1.2.2 Smart Contracts

Basic definition of smart contract as cited in [4]:

A smart contract is programming code that is stored and executed on the blockchain. Ethereum now has a Turing-complete language, Solidity, which enables developers to develop and deploy smart contracts. In addition to moving ether, the cryptocurrency in Ethereum network, between accounts, Ethereum smart contract code can support more modern program language constructs such as loops and perform much more complex computations, including data access, cryptographic algorithms, and function calls.[4]

A smart contract is like a scripted agreement between two or more interacting parties. The code built into the contract is stored on the Ethereum blockchain and once deployed, it cannot be changed or removed. This assures the credibility of the smart contract.

Typically, DApp developers write smart contracts for Ethereum blockchain in some high-level programming language and then compile them into the bytecode. The Ethereum bytecode is then deployed on the blockchain and is executed within the Ethereum Virtual Machine (EVM). The two most used languages used for Ethereum smart contract development are:

- **Solidity** is currently most commonly used object-oriented programming language for Ethereum smart contracts. It is influenced by C++ and Javascript and thanks to its extensive documentation is great for beginners. Solidity was created as a language explicitly for writing smart contracts with features to directly support execution in the decentralized environment of the Ethereum world computer.[14]

- **Vyper** is newer and more simple Python-like language that is however less developed, less documented than Solidity and is still considered experimental. Vyper code is compiled to the Application Binary Interface (ABI) and bytecode by Vyper compiler the same way Solidity does with Solidity compiler.[3]

As shown on example code in Figure 1.3, in terms of code length or readability, there is no clear winner as smart contracts don't usually contain highly complex algorithms.

Solidity

```
pragma solidity ^0.4.25;

contract OpenAuction {

    address public beneficiary;
    uint public auctionStart;
    uint public auctionStop;
    bool public ended;
    uint public highestBid;
    address public highestBidder;

    constructor (address _beneficiary, uint _biddingTime) public {
        beneficiary = _beneficiary;
        auctionStart = now;
        auctionStop = auctionStart + _biddingTime;
    }

    function bid() public payable {
        assert(now < auctionStop);
        assert(msg.value > highestBid);
        if (highestBid != 0) {
            highestBidder.transfer(highestBid);
        }
        highestBid = msg.value;
        highestBidder = msg.sender;
    }

    function endAuction() public {
        assert(now >= auctionStop);
        assert(!ended);
        ended = true;
        beneficiary.transfer(highestBid);
    }

}
```

Vyper

```
# Open Auction contract

beneficiary: public(address)
auctionStart: public(timestamp)
auctionStop: public(timestamp)
highestBid: public(wei_value)
highestBidder: public(address)
ended: public(bool)

# constructor
@public
def __init__(_beneficiary: address, _bidding_time: timedelta):
    self.beneficiary = _beneficiary
    self.auctionStart = block.timestamp
    self.auctionStop = self.auctionStart + _bidding_time

# create function for bidding
@public
@payable
def bid():
    assert block.timestamp < self.auctionStop
    assert msg.value > self.highestBid
    if not self.highestBid == 0:
        send(self.highestBidder, self.highestBid)
    self.highestBid = msg.value
    self.highestBidder = msg.sender

# end auction and send the highest bid to the beneficiary
@public
def endAuction():
    assert block.timestamp >= self.auctionStop
    assert not self.ended
    self.ended = True
    send(self.beneficiary, self.highestBid)
```

Figure 1.3: Solidity and Vyper comparison[3]

1.2.3 Oracles

Oracles definition as defined in [14]:

Oracles are systems that can provide external data sources to Ethereum smart contracts. In the context of blockchains, an oracle is a system that can answer questions that are external to Ethereum. Ideally oracles are systems that are trustless, meaning that they do not need to be trusted because they operate on decentralized principles.[14]

Oracles provide a way of getting off-chain information, such as the results of football games, the price of gold, or information about real estate, onto the Ethereum platform for smart contracts to use. They can also be used to relay data securely to DApp frontends directly.[14] Oracles can therefore be thought of as a mechanism for bridging the gap between the off-chain world and smart contracts. Allowing smart contracts to enforce contractual relationships based on real-world events and data broadens their scope dramatically.

1.2.4 Transactions and messages

In Ethereum, the term transaction represents the signed data package of a message that is sent from an Externally Owned Account (EOA) to another account. The message itself instructs what action to take on the blockchain. They all require the initiator of the transaction to digitally sign the messages, and transactions will be recorded into the blockchain.[4] Three types of transactions can happen:

- **Contract Account (CA) creation**: In this case, an EOA acts as the initiator or creator of the new contract account.
- **A transaction between two EOAs**: In this case, one EOA initiates an ether movement transaction by sending a message to the receiving EOA.
- **A transaction between EOA and CA**: In this case, the EOA initiates a message call transaction, and the CA will react with the referenced smart contract code execution.

The CA can send messages to other CAs or EOAs. Unlike the transaction, messages are virtual objects during the execution and will not be recorded into the blockchain.[4] If an EOA is the recipient, the recipient's account state will be updated and recorded in the world state. If a CA is the message recipient, they are accepted as function calls and the associated contract code will be executed.

1.2.5 Ethereum 2.0

The Ethereum that exists today is slow and expensive. The entire Ethereum network is throttled at 15 transactions per second and if complex processes are involved, the cost becomes astronomical[16]. One of the main reasons why Ethereum lacks in so many aspects comes down to one point: the high cost of decentralization.

These are some of the problems that Ethereum 2.0 aims to solve. Ethereum 2.0 will be able to process tens of thousands of transactions per second compared to the first iteration's 15[16]. Ethereum 2.0 also uses the Proof-of-Stake mechanism rather than the Proof-of-Work mechanism used by current Ethereum blockchain and Bitcoin.

1.2.6 Proof-of-Stake

View on Proof-of-Stake consensus as opposed to the Proof-of-Work one is described as follows:

> *As opposed to PoW consensus, where miners are rewarded for solving cryptographic puzzles, in the Proof-of-Stake (PoS) consensus algorithm, a pool of selected validators take turns proposing new blocks. The validator is chosen in a deterministic way, depending on its wealth, also defined as a stake. Anyone who deposits their coins as a stake can become a validator. The chance to participate may be proportional to the stakes they put in.*[4]

As demonstrated in the Figure 1.4, the blockchain keeps track of a set of validators, sometimes also called block creators or forgers. At any time, whenever new blocks need to be created, the blockchain randomly selects a validator. The selected validator verifies the transactions and proposes new blocks for all validators to agree on. New blocks are then voted on by all current validators. Voting power is based on the stake the validator puts in. Whoever proposes invalid transactions or blocks or votes maliciously, which means they intentionally compromise the integrity of the chain, may lose their stakes[4]. Upon the new blocks being accepted, the block creator can collect the transaction fee as the reward for the work of creating new blocks.

Overall, PoS is much more energy-efficient and environment-friendly compared with the PoW mechanism. It is also perceived as more secure too. [4] It essentially reduces the threat of a 51% attack since malicious validators would need to accumulate more than 50% of the total stakes in order to take over the blockchain network whereas in Bitcoin case attacker needs more than 50% of hash power which – while still very difficult – is easier to achieve.

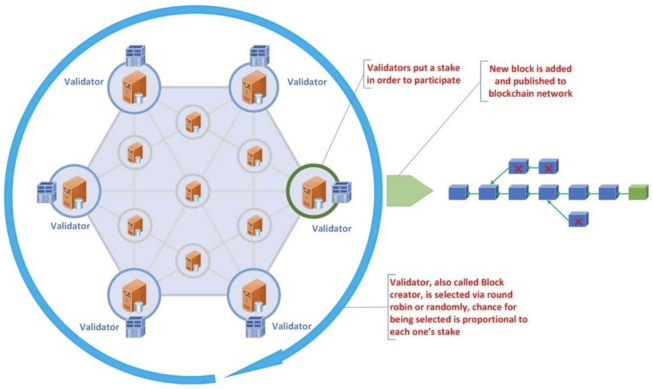

Figure 1.4: Proof-of-Stake visualization[4]

1.3 DasContract

DasContract[5] is a working concept of model used for generating smart contracts for different blockchains. It is currently still in development and was introduced by Ing. Marek Skotnica and doc. Ing. Robert Pergl, Ph.D. from CTU in Prague.

The DasContract is trying to solve problems with agreements, mainly legal texts, which may contain errors and cause ambiguous interpretation. With this type of ontological contract, rules are forced to be obeyed and ambiguity is controlled as opposed to other currently commonly used types of agreements shown by contract maturity model on Figure 1.5.

Maturity	Name	Contract Form	Accuracy
1	Verbal contract	A mutual understanding	No written record of a contract
2	Written informal contract	Informal text	Typically ambiguous interpretation, possible errors, no legal framework
3	Legally binding contract	Legal text	Risks of ambiguous interpretation, possible errors, legal framework contains ambiguities itself
4	Ontological contract	Ontological model	Ambiguity effectively controlled

Figure 1.5: A contract maturity model[5]

As said in the paper and shown on Figure 1.6, DasContract's proposed approach consists of three parts:

Human Understanding *part defines a contract between multiple parties that they need to agree on. Such a contract is a combination of legal text and formal ontological models. The legal text in some form specifies the legal validity of the formal model. The formal models need to be unambiguous, so only one possible interpretation is allowed.*

Technical Implementation *part specifies how formal models from the contract are transformed into a software executable code and uploaded into a blockchain as a smart contract.*

Digital Interaction *is a part where people, companies and legal authorities can interact with the agreed upon contracts. Since the contract is in a blockchain, the interaction is fully digital, and thanks to cryptography can also be legally binding. Blockchain by design also provides an audit trail of all actions performed by the parties and ensures that the agreed upon contract is executed correctly.[5]*

1.3.1 DasContract structure

DasContract consists of two main parts:

- **Process Model** models the flow and states of the contract. Process model defines activities which are providing logic to the states used in process model .

- **Data Model** defines entities and their properties used in the contract.

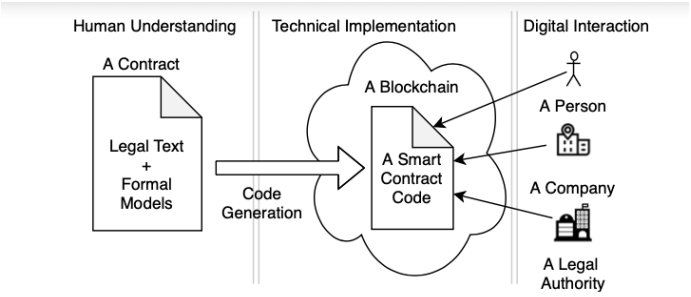

Figure 1.6: A proposed concept architecture used by DasContract[5]

1.3.1.1 DasContract file structure

Generated file with .dascontract extension is XML file consisting of contract's ID, name and processes. Each process is further divided into:

- **Diagram** for saving the original process model, commonly but not limited to as BPMN format. Other example could be XML generated from Blockly graphical language *https://developers.google.com/blockly*.
- **Sequence flows** connecting activities and gateways with source and destination attributes.
- **Process elements** including activities and gateways.
- **Entities** with properties as representation of contract's data model.

1.3.2 DasContract designer

DasContract can be – as of writing this paper – generated by DasContract Editor[17] found on *https://dascontracteditor.azurewebsites.net/*. This editor uses BPMN diagram for its process model part of contract, currently implementing start and end events, sequence flows, three types of tasks and two types of gateways:

- **User task** converts into smart contract's public function that can be executed either by designated or any user of the blockchain, depending on contract's design. At the time of writing the user task can't have any other executable code than proceeding to next object in sequence flow and mapping its parameters to data model, however custom smart contract code part is planned by DasContract developers as one of the next features.
- **Script task** is used for the code that is specific for the contract and can't be automatically generated. This may include transferring specific sum of money to some account or making custom logic inside smart contract.

Script task converts into private functions that can't be executed by anyone and it's execution depends completely on sequence flow.

- **Business task** is using decision table to transform input to output.
- **Exclusive gateway** inherits its functionality from BPMN model and chooses the path based on conditions of sequence flows going out from the gateway.
- **Parallel gateway** allows executing multiple tasks independently on each other. As opposed to its BPMN implementation, in DasContract parallel gateways work a little bit differently and don't need to be joined into one thread.

Example of primitive contract using user task, script task and parallel gateway is shown on Figrue 1.7. It is very simplified example for demonstrative purposes only and by no means should be used in production solution since it has some security flaws. This scenario works as escrow service for selling items that are represented by non-fungible token on blockchain. Right after start event follows parallel gateway which splits program to two subprocesses which both have to completely execute to continue with the program after joining in the second gateway. There are two user tasks where seller sends the token to escrow (to the contract address) and buyer pays for the item and sends given amount of money (e.g. tokens backed up by USD like Tether) also to the escrow. Finally, when both of these events are completed, the last task is processed and since it is script task, it automatically execute given code and sends money to seller and the item to buyer.

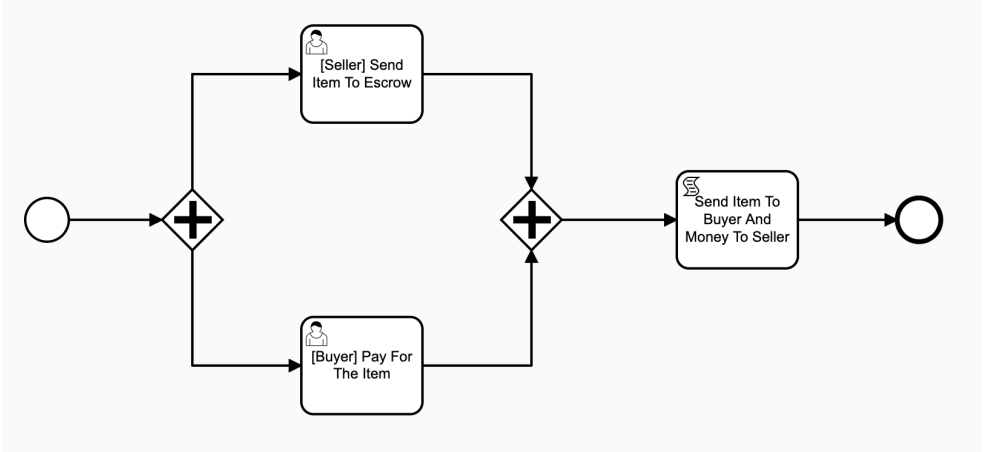

Figure 1.7: Example process model of DasContract using DasContract designer

1.3.2.1 BPMN

Slightly modified BPMN notation is heavily used in the DasContract editor and is currently one of the main process modelling notations used by DasContract itself. The following paragraph describes the BPMN language: *The Business Process Modeling Notation (BPMN) is visual modeling language for business analysis applications and specifying enterprise process workflows, which is an open standard notation for graphical flowcharts that is used to define business process workflows. It is popular and intuitive graphic that can be easily understand by all business stakeholders, including business users, business analysts, software developers, and data architects.[18]*

1.3.3 Comparison with Caterpillar

There aren't many competitors, however project called Caterpillar is very similar and can be found on *https://github.com/orlenyslp/Caterpillar*. Caterpillar uses TypeScript as programming language opposed to DasContract's smart contract generator that is written in C#.

Caterpillar doesn't share complexity DasContract's solution and is available only for BPMN to Ethereum smart contract conversion instead of multiple choice of input graphical language and multiple choice of output blockchain solutions for smart contract. It is however much more developed and with its features exceeds DasContract in the current form.

Caterpillar's **advantages** are: REST API, currently better support for BPMN-to-Solidity conversion and better documentation.[19]

Disadvantages include single choice of BPMN-to-Solidity conversion while DasContract supports multiple choices on both sides and non-active development – at the time of writing the last major update was 14 months ago and last commit 7 months ago, while DasContract is still being actively developed.

1.4 Chapter summary

This chapter consisted of some theoretical background that is good to know for understanding this paper and implementation of practical part of thesis.

First part was dedicated to blockchain as a whole, the history, idea behind it, cryptographic techniques, principle and consensus algorithms. Bitcoin as the first blockchain implementation and important part of the history was also more discussed as were other popular blockchains and their use cases.

The second section was about Ethereum, the blockchain which this thesis uses as basis for its practical part, where Solidity code – the Ethereum's smart contract language – is generated. Other topics discussed include tokens in Ethereum, Oracles as source of off-chain information or the soon coming Ethereum 2.0.

Last part described DasContract from which are the smart contracts in practical part generated. We were looking to idea behind it, structure and components as well as similar project Caterpillar which we have compared DasContract to.

Chapter 2

Evaluation of Ethereum strengths and weaknesses

In Chapter 2 we will look to different generations of blockchain projects, what is defining for each one and which is currently the best for use case of this thesis. Next we assess strengths and weaknesses of Ethereum blockchain and make conclusion out of them.

2.1 Three generations of blockchain

As said in Chapter 1, large amounts of different implementations of blockchain were developed since Bitcoin whitepaper was released. The community distinguish between three generations and before we get into evaluation of Ethereum itself, let's compare differences between generations and decide why the second one is in the current situation the best option for writing semi-production-ready smart contracts:

- **First generation** allows simple transactions like token transfer that brought to the general public practical example of how blockchain works. Examples are Bitcoin or Litecoin and its tokens (also known as cryptocurrencies) should have been used mainly for payment as decentralized and unregulated currency. The first generation is here for more than a decade and is currently very stable and with minimum of bugs and security threats.

- **Second generation** is where blockchains started to orientate more to the way of smart contracts that should (among other use cases) support current monetary systems around the world, making them better, faster and more secure, instead of replacing it. The tokens are now more often representing real life objects and money as opposed to trying to replace

them. The most known representative of second generation is Ethereum, released to the public in 2015.

- The main features of the **third generation** are wider functionality and better design that allows avoiding such problems as poor scalability. Another feature common for the third generation of blockchains is the ability to process crosschain transactions. The rest notable features are inbuilt compliance and governance and improved mechanism of smart contracts (inbuilt formal software verification). The Proof-of-Work is usually replaced by other consensus mechanisms (e.g. Proof-of-Stake or Proof-of-Authority).[20] Most known examples of the third generation are Cardano, EOS or VeChain, all mentioned in section 1.1.5.

Overall the third generation looks very promising and definitely as an improvement to the second one. However blockchains included in this generation are very young and immature, large portion of them aren't even completed and production ready and even those that are in use still have very recent security breaches, e.g. VeChain's lost of tokens worth $6.6M in December 2019.[21]

On the other hand Ethereum is years without major breaches. And while still not completely ideal because of relatively high fees for transactions and Proof-of-Work consensus, it is currently the best compromise between security and functionality. That being said, in Ethereum 2.0 update, Ethereum will switch to Proof-of-Stake consensus algorithm making it more secure, cheaper and faster while – hopefully – keeping its reliability.

2.2 Ethereum's strengths

Majority of blockchains allow the smart contracts which is all we need for DasContract's output. There are however few features and qualities that make Ethereum the better option. These include:

- **Reliability** - Ethereum is running on production version for 5 years and at time of writing is 2nd most valued cryptocurrency (behind Bitcoin) with $29.5B. High value of blockchain of course isn't necessarily benefit, although it represents the trust of reliability and stability of the blockchain that other people and companies have in Ethereum.

- **Security** - As stated in section 2.1, Ethereum is in its current state very secure with no breaches in last years. Of course Ethereum still uses Proof-of-Work consensus algorithm that is vulnerable to 51% attack, where single person or co-working group of people own more than 50% of hashing power on the blockchain and may then authorize fake transactions. This type of attack is currently still possible in Ethereum, however with great difficulty. According to *https://www.crypto51.app/*

1 hour of attack would theoretically cost almost $200,000.[22] This may not sound like a large sum of money for the ability to attack $29.5B blockchain and it indeed isn't. There is second problem - on the largest platform for buying hash power, there is currently only 4% of required power available for renting and buying dedicated hardware would cost tens of millions of dollars.

- **Decentralization** - one of the advantages of Proof-of-Work algorithm is high decentralization – if the blockchain is popular and rewards are attractive for people mining new blocks. This is the case of Ethereum and because of that, nodes are located all over the world. Majority of them are in Europe and North America, on the other hand some may be found in countries like Kenya, Mauritius or Ghana [23]. The decentralization is important to ensure no political decision, natural disaster or other disturbtion in one part of world would not make blockchain unusable.
- **Solidity, developer base, documentation** - Even the flawless, best possible blockchain would be no good for real world usage without smart apps, their developers and users using them. This is where Ethereum has the edge over any other blockchain. Even though is slower than newer third generation ones with Proof-of-Stake consensus, Ethereum runs much more decentralized apps than any other public blockchain. Users of the apps don't even have to know on which blockchain is the app running or how it is implemented. This decision is on developers and they usually tend to go easier and proven way of well documented platform such as Ethereum/Solidity tandem, especially when the app should be used in production.
- **Ethereum 2.0** - With 2.0 update comes significant change from old Proof-of-Work consensus algorithm to the newer and universally acclaimed as better one Proof-of-Stake. Thanks to this change, Ethereum will dramatically increase network bandwidth and reduce gas cost per transaction meaning transactions will become faster and cheaper. On top of this user benefit, security of the Ethereum will increase. The 51% attack on Proof-of-Stake blockchain would require holding more than 50% of all ether tokens, which would be with current prices around $14.75B. In essence, 2.0 update will upgrade Ethereum to the third generation of blockchains.

2.3 Ethereum's weaknesses

- **Proof-of-Work** - Although Proof-of-Work consensus will be abandoned by Ethereum soon, there might be delays caused by failures in testing networks and we need to still count with Proof-of-Work as current

consensus algorithm with its weaknesses that were already mentioned including bad scalability, high transaction fees or easily achieved 51% attack as opposed to Proof-of-Stake.

- **High transaction fees** - With every transaction either to transfer tokens or to execute smart contract, gas has to be sent along with the transaction. The fees go to block creators as reward for being part of the network and producing new blocks. With Proof-of-Work algorithm, Ethereum achieves around 15 transactions per second (TPS), however Ethereum 2.0 Proof-of-Stake on testing network achieves around 7 000 TPS with vision of 100 000 TPS in 2-year horizon.[24] Rewards will on the other hand remain very similar and transaction fees will then be up to 5000 times lower. Currently the fees are around $0.36 per transaction when the network isn't saturated.[25] In peaks, fees may exceed $1 and while for some large currency transfers this fee is completely fine, for smart applications using smart contracts the fee is not acceptable and would not be profitable in production.
- **Ethereum 2.0** - The new update will be definitely advantage, if everything goes well. Developers of Ethereum are deploying public testing networks and they are making sure everything goes smoothly. If it for any reason does not however, they will have big problem. And that counts not only for the transition itself, but even for some security errors that might be in code of update. Current version is stable and secure, but what if the 2.0 after launch won't be? Then a lot of users and developers might be searching for alternative and it could be very hard obstacle to overcome for Ethereum.
- **Other blockchains** - This isn't directly Ethereum's fault, however as time goes by, new, more powerful blockchains are released and community around them is making them better and more secure. They all have one major advantage – don't have to assure backwards compatibility and smooth transition from one version to another like Ethereum has to. This means the development of such blockchains is faster, cheaper and they can come with some revolutionary ideas, e.g. connection of AI and blockchain. Such blockchain would have great competitive advantage over Ethereum and developer might transfer over to the newer and better one.

2.4 Chapter summary

Ethereum is one of the most mature public blockchains out there. It has a lot of advantages over competitors, mostly coming from the popularity. There are however some functionalities that could be improved and Ethereum developers are addressing them and will try to improve them in the next major

update Ethereum 2.0 coming later in 2020.

To sum up and for clarity follows Table 2.1 with strengths and weaknesses of Ethereum blockchain:

Strengths	**Weaknesses**
Reliability	Proof-of-Work consensus
Security	High transaction fees
Decentralization	Ethereum 2.0 (possible errors)
Documentation	Other competitive blockchains
Ethereum 2.0 (PoS consensus)	

Table 2.1: Ethereum strengths and weaknesses

Chapter 3

Analysis and design

In this chapter we will look deeper into Solidity code, explain some specific parts of the language. We will explain how exactly the DasContract code is converted to them using .NET Core platform and C# language. Those parts include different variable types, global variables, modifiers, functions, their visibility and linking them together with sequence flows in Ethereum smart contract code.

3.1 Design

3.1.1 Implementation requirements

In .NET Core framework implement program that will convert given DasContract file to Solidity smart contract code. The solution should include unit tests as separate project and the Solidity output should be be manually tested and debugged in Remix[26] IDE.

3.1.2 Class diagram

Proposed class diagram shown on Figure 3.1 consists of three parts. The `ProcessElement` class is already implemented as an abstraction of the DasContract. This thesis therefore develops the other two abstract classes and their subclasses. `SolidityComponent` classes solve transformation of the initialized class objects to Solidity code. `ElementConverter` classes are converting objects from DasContract abstraction by taking `ProcessElement` as constructor parameter and returning initialized `SolidityComponent` with all parameters from the DasContract object.

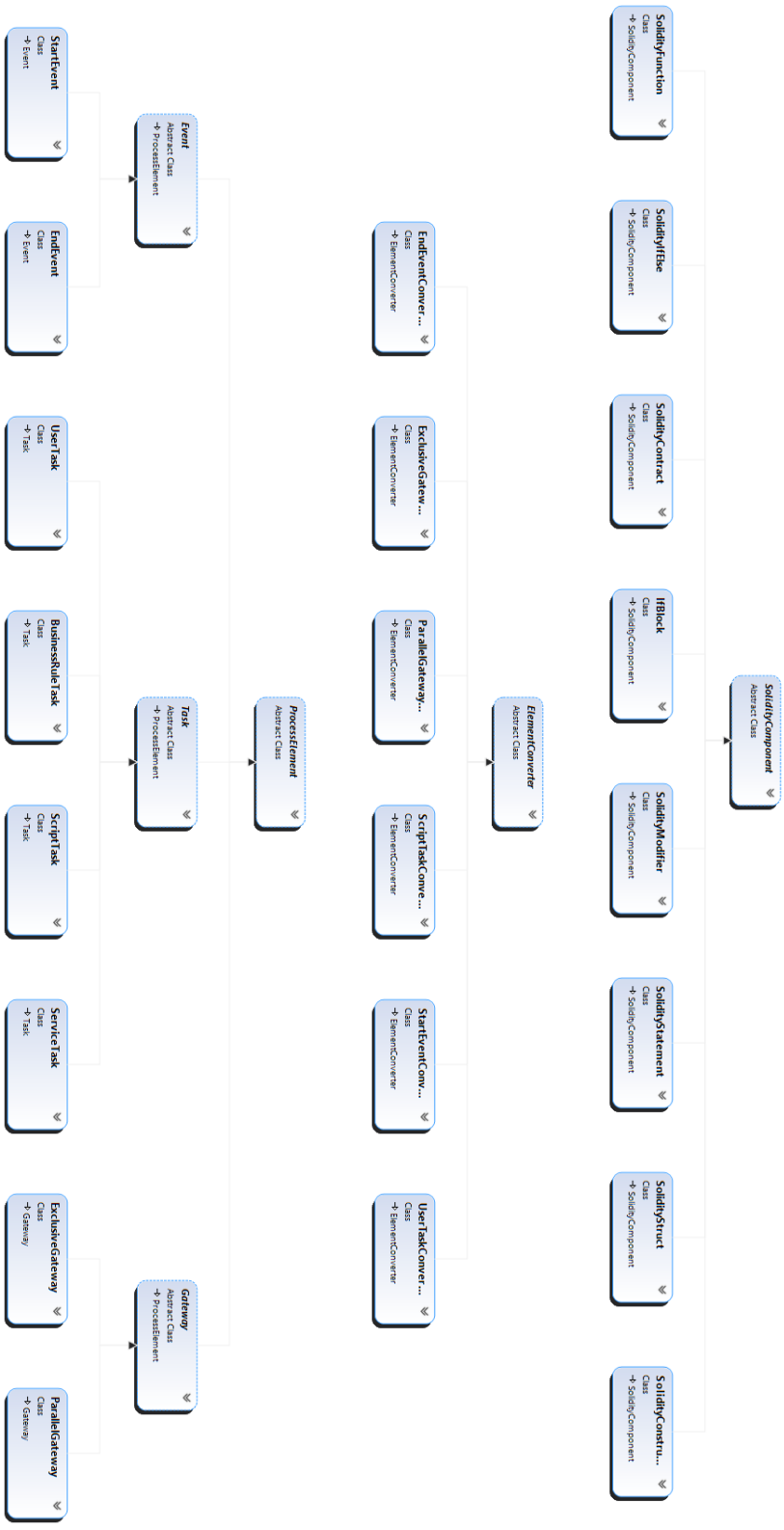

Figure 3.1: Class diagram of the program

3.2 Code generating from DasContract to Solidity

We will now explain the logic of program in the practical part and how it converts simple DasContract file to the Solidity code of smart contract.

Each part of Solidity language generation will be demonstrated on a snippet of code from our testing DasContract file which diagram is shown at Figure 1.7. The DasContract code is generated by DasContract Editor[17].

3.2.1 Solidity file structure

3.2.1.1 Pragmas

Every Solidity file starts with with one or more pragma keywords[27] that are used to enable certain compiler features or checks.

The first and most important pragma keyword is the one declaring version of Solidity language that should be used and is mandatory. The version pragma is used as follows: `pragma solidity ^0.6.6;`. This statement means the code will only be compiled with any Solidity compiler version lower than `0.6.6`. However the `^` symbol allows to use higher compiler version `0.6.x` where `x >= 6` since these are only minor language changes and shouldn't break code compilation. This won't however enable versions `0.7.0` upwards.

In DasContract, there isn't currently implementation of Solidity version directly in model and has to be defined directly in code generator program or changed in generated Solidity code.

Experimental pragmas are still in the experimental stage and aren't currently supported by DasContract, although they may be implemented at later time so we will list them to be complete:

- `pragma experimental ABIEncoderV2` is used for decoding nested structures in Solidity [27]. This also allows usage of structure as parameter of non-public functions.

- `pragma experimental SMTChecker` can find safety warnings in code as protection from attackers, e.g. when using inappropriate variable types, and list them as warnings.

3.2.1.2 Imports

Importing source code from another files is standard across vast majority of programming languages and Solidity is no exception. Same as pragmas, imports are not handled by DasContract model and they need to be directly defined in the generator program or added afterwards in the generated Solidity code.

Imports are available for both local offline importing (`import filename`) and online storage, mostly used for importing ERC token definitions, e.g.

for using ERC-721 token interface `import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v3.1.0/contracts/token/ERC721/IERC721.sol";`.

3.2.2 Variables

Solidity variables are in the program generated from different parts of DasContract code. Most of them are generated from user defined data model directly into structures. Then there are internal variables which users don't directly interact with, instead they are helping to remember the contract state. These include integer counters for parallel gateways and arrays for storing active states and address mappings from user's names to their addresses.

3.2.2.1 Global variables

Solidity defines large number of global variables and properties that can be used without declaration in any part of smart contract. We won't be listing all of them, only the ones important for current DasContract state and the Solidity code generator implementation.

Message variable `msg` refers to the current transaction that is being processed by smart contract and defines several properties:

- **`msg.value`** defines the amount of ether sent with the transaction. This property is especially useful with `payable` methods, allowing claim some or all of the sent amount to the smart contract.
- **`msg.sender`** stores the address of the transaction sender. This address is stored in the generated smart contract to the mapping of user names that are optionally defined in DasContract's activities.

The `gasleft()` method returns amount of gas remaining for transaction execution. When the gas reaches 0, the transaction is automatically reverted and not stored in blockchain without needed to use the function. It can be however used for transactions that we know in advance to require large amount of gas and we don't want the sender to spend all the gas when the transaction would be reverted. We can then use `require()` statement and stop the transaction right away. E.g. if we know that the transaction will consume more than 300 000 gas, we can use `require (gasleft() >= 300 000)` statement. This way the transactor will lose only the gas needed to process this one function instead of all of 300 000 gas.

3.2.2.2 Variable types

There are different types of variables, mostly the same as in other languages. In this section, we will take a look at atypical variables that are present only in Solidity and other smart contract languages or at the ones that are similar

to typical variables, however used differently.

`Address` as the name suggests holds 20 byte value, exactly the size of Ethereum address[27]. The `address` variable implements three functions, the only important one for our purposes is `address.balance()`. It is public function, allowing anyone on the blockchain check addresses balances of both ether and ERC tokens.

`Address payable` is extension of the simple `address` variable. The only difference is implementation of another two functions – `address.send()` and `address.transfer()`. Both functions are used to send ether from one Ethereum address to another. The only difference is `address.send()` returning `false` on failure and the `address.transfer()` reverts the whole transaction. Both are perfectly usable and depends on use case of the contract function.

The `address (payable)` variables are both included in DasContract code as shown in this code snippet:

```
<PrimitiveContractProperty>
  <Name>Sender</Name>
  <Type>AddressPayable</Type>
</PrimitiveContractProperty>
```

This code transfers to `address payable Sender = address(0x0);` in the final Solidity code generated by the program.

Structures are supported by Solidity language and heavily used in generated Solidity code. They have however some limitations in contrast to other languages. Structures in Solidity:

- can't contain a member of its own type because Solidity code has to ensure limited size of the structure and it isn't possible with possibility of infinite structure nesting.
- can't be used as parameters in functions by default. They may be used as parameters with usage of experimental pragma `ABIEncoderV2`, even then there are limitations such as the structure can't contain mapping datatype.
- can't contain any methods or method references.

Now let's take a look at example of one entity from our example. This will be the `Item` entity with primitive properties `ItemPrice`, `Name`, `ItemTokenID` and reference property `Payment` which is another entity in our DasContract. The `Item` entity follows:

```
<ContractEntity>
  <Id>c59258cb-1147-4c8f-8951-d162e31e4ade</Id>
```

```
    <Name>Item</Name>
    <PrimitiveProperties>
      <PrimitiveContractProperty>
        <Id>28b4b696-b6db-45ea-afb2-9f036f95f3e3</Id>
        <Name>ItemTokenID</Name>
        <IsMandatory>true</IsMandatory>
        <Type>Number</Type>
      </PrimitiveContractProperty>
      <PrimitiveContractProperty>
        <Id>3aad96f6-123e-417f-8081-56122d27264a</Id>
        <Name>ItemPrice</Name>
        <IsMandatory>true</IsMandatory>
        <Type>Number</Type>
      </PrimitiveContractProperty>
      <PrimitiveContractProperty>
        <Id>8bce912a-0052-4445-8a48-a2d7448df742</Id>
        <Name>Name</Name>
        <IsMandatory>true</IsMandatory>
        <Type>Text</Type>
      </PrimitiveContractProperty>
    </PrimitiveProperties>
    <ReferenceProperties>
      <ReferenceContractProperty>
        <Id>0740074a-c8a3-4394-ad18-a7ba812c5b57</Id>
        <Name>Payment</Name>
        <IsMandatory>true</IsMandatory>
        <EntityId>213b58fa-1c84-4343-9250-70050f6a972b
        </EntityId>
        <Type>SingleReference</Type>
      </ReferenceContractProperty>
    </ReferenceProperties>
  </ContractEntity>
```

Each entity begins with `ContractEntity` XML element followed by `Id` which is used in the transformation program as internal identifier that is however not transferred to the Solidity code, where is the structure's name is defined by `Name` element's value.

Next follows `PrimitiveProperties` and `ReferenceProperties` elements each containing one or more `PrimitiveContractProperty` and `ReferenceContractProperty`, respectively. Each of them contain elements `Id` for identification, `Name` used as variable name and `IsMandatory` which isn't used by converter and will be used for another parts of the whole DasContract project.

The differences between primitive and reference properties are in types. While primitive property defines standard variable types like `Text` (`string`

in Solidity code) or `Number` (`uint256` in Solidity code), reference property defines type by combining `Type` and `EntityId` elements. `Type` can be either `SingleReference` for simple variable or `CollectionOfReferences` for array. The `EntityId` then defines which structure should be used as the variable type. In this example value of `EntityId` refers to `Id` of `Payment` entity.

After defining the structure members we need to initialize variable of the given structure. The name of generated structure in Solidity code always begins with capital letter and the variable has the same name except the first letter is small. To initialize structure, we need to know the default values – default values of primitive data types are defined in our generator. For reference variable we have to initialize the reference structure `Payment` inside our defined structure `Item`.
Finally, the generated code looks like this:

```
  struct Item{
        uint256 itemTokenID;
        uint256 itemPrice;
        string name;
        Payment payment;
}
Item item = Item({itemTokenID: 0, itemPrice: 0, name: "",
payment: Payment({sender: address(0x0), amount: 0})});
```

Mapping types are declared as `mapping(_KeyType => _ValueType)`[27]. `_KeyType` can be any primitive data type, while `_ValueType` may also include another mapping or structure. Mappings are basically hash tables where every possible key exist and are virtually initialized to the default value of the `_ValueType`.

Mappings don't have any iterator or way to get all initialized keys. That means that for most use cases it is needed to remember keys vector. This way can be implemented automatically expanding arrays with the mapping itself being array and helper integer that stores last index of array as implemented in the Solidity code generator.

When generating smart contract from DasContract file, there are always at least two mappings shown below. The `addressMapping` is used for storing specific address to address identifier from user tasks in DasContract. The `activeStates` mapping stores states and their status (`true` or `false`) whether they can be executed or not.

Mappings example from generated Solidity code:

```
mapping (string => address) public addressMapping;
mapping (string => bool) public activeStates;
```

3.2.3 Functions

The majority of generated code from DasContract to Solidity are functions and modifiers. Firstly we will go through functions.

There are several parts of DasContract code from which are functions generated. First of them is constructor function:

```
<ContractProcessElement xsi:type="ContractStartEvent">
    <Id>StartEvent_1nygn6s</Id>
    <Incoming />
    <Outgoing>
        <string>Flow_0854rbi</string>
    </Outgoing>
    <StartForm>
        <Id>19e60caf-638e-4cc4-9a5c-49931ac26397</Id>
        <Fields />
    </StartForm>
</ContractProcessElement>
```

As seen from this example, **start events** are the only ones without incoming sequence flows and are defined by `xsi:type="ContractStartEvent"` attribute. DasContract is currently limited to only one start event and the function generated from start event is constructor function. This constructor executes when the smart contract is deployed to the Ethereum blockchain and does only one thing – activates the next element in the flow.

Generated Solidity example from code above:

```
constructor () public payable{
        Gateway_1mcm5twLogic();
}
```

The next type of generated functions are **gateway** functions. In DasContract, there are two types of gateways, exclusive and parallel. In the example contract, the parallel gateway is implemented as follows:

```
<ContractProcessElement
xsi:type="ContractParallelGateway">
  <Id>Gateway_17ps7az</Id>
  <Incoming>
    <string>Flow_146x4ic</string>
    <string>Flow_05upkfb</string>
  </Incoming>
  <Outgoing>
    <string>Flow_09gz9ue</string>
  </Outgoing>
</ContractProcessElement>
```

The converter logic looks to the number of elements inside `Incoming` and `Outgoing`. The generated function of parallel gateway logic activates all `Outgoing` flows. If there are multiple `Incoming` flows, the counter for the gateway is generated to the Solidity code, in this example the counter is defined at the top of smart contract as `int Gateway_17ps7azIncoming = 0;` and it increases every time the flow reaches the gateway. When the counter value is equal to the number of flows incoming to the gateway, only then the gateway logic is executed.

The parallel gateway Solidity code from the example ("SendItemToBuyerandMoneyToSeller" activity name is replaced by "SITBAMTS" for the document formatting reasons):

```
function Gateway_17ps7azLogic() internal {
        if(Gateway_17ps7azIncoming==2){
                ActiveStates["SITBAMTS"] = true;
                SendItemToBuyerandMoneyToSeller();
                Gateway_17ps7azIncoming = 0;
        }
}
```

The exclusive gateway isn't represented in the example contract. Converter program takes the outgoing flows and based on conditions – stored as outgoing flows names – makes decision which of the next element's functions to activate. Thanks to its exclusivity, there is no need for counter like in the parallel gateway and the function of gateway logic will execute immediately first time it activates.

Last implementations of functions in the DasContract are activities. User activities can accept parameters and save them to contract data model using property binding logic. The XML code snippet of `PayForTheItem` user activity (`PropertyId`s shortened):

```
<ContractProcessElement xsi:type="ContractUserActivity">
  <Id>Activity_07vsk5o</Id>
  <Name>[Buyer] Pay For The Item</Name>
  <Incoming>
    <string>Flow_0loskjc</string>
  </Incoming>
  <Outgoing>
    <string>Flow_05upkfb</string>
  </Outgoing>
  <Form>
    <Id>7664f788-9fda-426a-a806-40cf7a974d03</Id>
    <Fields>
      <ContractFormField>
```

```
            <Id>143233d4-5190-4aa3-9645-d4ba2b7f1916</Id>
            <Name>Sender</Name>
            <Label>Sender</Label>
            <ReadOnly>false</ReadOnly>
            <PropertyBinding>
              <PropertyId>ecae65e5-70e0-4803</PropertyId>
            </PropertyBinding>
          </ContractFormField>
          <ContractFormField>
            <Id>d338872c-e65e-4a0c-a6bd-bcb47882f6c9</Id>
            <Name>Amount</Name>
            <Label>Amount</Label>
            <ReadOnly>false</ReadOnly>
            <PropertyBinding>
              <PropertyId>512cc04a-c282-4759</PropertyId>
            </PropertyBinding>
          </ContractFormField>
        </Fields>
      </Form>
</ContractProcessElement>
```

Same way as gateway functions, activity functions have incoming and outgoing sequence flows, only this time there must be exactly one incoming and one outgoing flow.

The `ContractFormField` elements contain information about function parameters and their binding to the string instances created from data model. The important elements include the `Name` element defining parameter name and `PropertyBinding` which defines parameter type and variable where to save the parameter value. This is done in the generator logic using `PropertyId` which is matched with the actual variable.

Another important feature of the activity is its name prefix in parenthesis, in this example `[Buyer]`. This is optional feature enabling function execution to only one address. The first time in the flow, any address can execute such function, however next time some function has the same prefix name, only the address used to transact the first function with that prefix can be used.

The Solidity generated code of the example `PayForTheItem` function (formatting slightly changed):

```
function PayForTheItem
(address payable Sender, uint256 Amount)
isPayForTheItemState isPayForTheItemAuthorized public {
        ActiveStates["PayForTheItem"] = false;
        payment.sender = Sender;
        payment.amount = Amount;
```

```
        Gateway_17ps7azIncoming += 1;
        Gateway_17ps7azLogic();
}
```

The script activities are always internal or private, hence non-transactable directly by blockchain users. Activities are instead run when the sequence flow reaches them, usually after user transact the user activity.

Each script activity is empty by default, only with changing active states in the `activeStates` mapping. The actual logic has to be added manually in the DasContract editor. When generating the Solidity code, script is added to the body of function generated from the activity.

We will now look at the DasContract XML snippet (slightly formatted):

```
<ContractProcessElement
xsi:type="ContractScriptActivity">
  <Id>Activity_16sr07p</Id>
  <Name>Send Item To Buyer and Money To Seller</Name>
  <Incoming>
    <string>Flow_09gz9ue</string>
  </Incoming>
  <Outgoing>
    <string>Flow_1un501e</string>
  </Outgoing>
  <Script>transferItemAndMoney();</Script>
</ContractProcessElement>
```

The code is the same as in user activity case with two exceptions. Script function doesn't accept any parameters and it has `Script` element containing the script that will execute with the function. In this example it is only one line function, it can however be as long as needed.

This XML will translate to the Solidity code below (slightly formatted):

```
function SendItemToBuyerandMoneyToSeller()
isSendItemToBuyerandMoneyToSellerState internal {
        ActiveStates["SITBAMTS"] = false;
        transferItemAndMoney();
        ActiveStates["Event_1nfb3wk"] = true;
}
```

3.2.3.1 Visibility

Solidity distinguishes between four types of visibility: `external`, `public`, `internal` and `private`. The Solidity code generator although uses only the

`public` and `internal` ones.

`Internal` functions may be transacted only internally and by contracts deriving from the one which is implementing them.

`Public` functions can be transacted by any other smart contract as well as by messages sent by users (addresses). It theoretically creates issue where users can skip functions in the flow. Every function – even internal – is however guarded by modifier allowing only active functions to be executed and the other ones are automatically reverted.

3.2.3.2 Modifiers

Function modifiers in Solidity can be used to change the behaviour of functions in a declarative way [27]. They can add functionality to the function or restrict access to them by checking condition and if not fulfilled, the function will revert to the previous state. In the generated code, three different modifiers are being used.

The first one is `payable` modifier that allows the function to accept payments in ether. The ether amount is sent together with message and differs from the gas sent. Receiver will always receive the full amount of sent ether, fees are paid from the gas and if there isn't enough gas, the transaction will revert instead of paying rest of the fee from ether sent with transaction. Every `payable` function has to be public since addresses have to be able to send ether to the smart contract via these functions.

The other two modifiers used in generated Solidity codes are defined in the code and are also generated.

One of them is responsible for checking whether the state is active by looking up the function name in `ActiveStates` mapping. Thanks to this modifier the sequence flow can't be hacked and only the state(s) next in the flow may be executed. Name of such modifier is derived from function name with prefix `is-` and suffix `-State`.

Example of generated `is-State` modifier:

```
modifier isPayForTheItemState{
        require(isStateActive("PayForTheItem")==true);
        _;
}
```

The second of custom modifiers checks whether the transacting address is authorized to transact the function. It checks whether some address is already mapped to the alias used in square brackets of the activity name and if so, only this address can transact given function. If on the other hand no address is assigned (marked as default address value `0x0`), the currently transacting address is mapped to the alias name. Name of this modifier is also derived from function name with prefix `is-` and suffix `-Authorized`.

Example of generated `is-Authorized` modifier:

```
modifier isPayForTheItemAuthorized{
        if(addressMapping["Buyer"]==address(0x0)){
                addressMapping["Buyer"] = msg.sender;
        }
        require(msg.sender==addressMapping["Buyer"]);
        _;
}
```

3.2.4 Error handling

Solidity provides two basic functions for error handling – `assert` and `require`. Both functions accept boolean expression as parameter and throw exception when the expression evaluates as `false`. The generator uses only `require` function as it is recommended to use for user input errors and state checks[27]. The `assert` function should be only used to test for internal errors.

The generator is only using `require` function in modifiers bodies as demonstrated on example codes above in `is-State` and `is-Authorized` modifiers. Error checking functions are however not limited to usage only in modifier and can be also used in function body, e.g. in script of script activity.

3.3 Chapter summary

This chapter described process of transformation code from DasContract file format to Solidity smart contract code.

Chapter also consisted of some of the Solidity language similarities and differences from standard programming languages and described the challenges that come with it and how it is solved by the program that was developed as part of this thesis.

Finally, we went through both XML and Solidity code of the item escrow example project and discussed the conversion process.

Chapter 4

Implementation

Chapter 4 will discuss the implementation of program from practical part starting from used technologies and code generating flow, to the Proof-of-Concept implementation, the example usage of DasContract solution with mortgage use case included with the program. We will take a look at selected specific activities as well as flow of the contract as a whole. Last topic included in this chapter are two different approaches to testing that were used, unit testing and manual testing of the generated Solidity code.

4.1 Used technologies

Essential technologies and tools used for the practical part of this thesis include:

- `.NET Core`, a free cross platform framework, successor to .NET Framework. The entire code generator from DasContract to Solidity was written in C# language using the .NET Core 3.1 framework.
- `Remix` found on `https://remix.ethereum.org/` is a tool for running and testing smart contracts outside of blockchain. It was very helpful when testing generated smart contracts as it doesn't require any setup of own blockchain and runs locally in web browser[26]. Remix also outputs compilation errors and warnings as well as implements decent debugger while the smart contract is deployed.

4.2 Code generating flow

So far we have discussed how the individual parts of code are generated from the DasContract file. In this section we will take a look at how exactly the program flow proceeds.

The program structure respects contract diagram as shown on Figure 4.1 and starts with `Contract` class which is filled with all entities, flows and their attributes by XML parsing of the DasContract file.

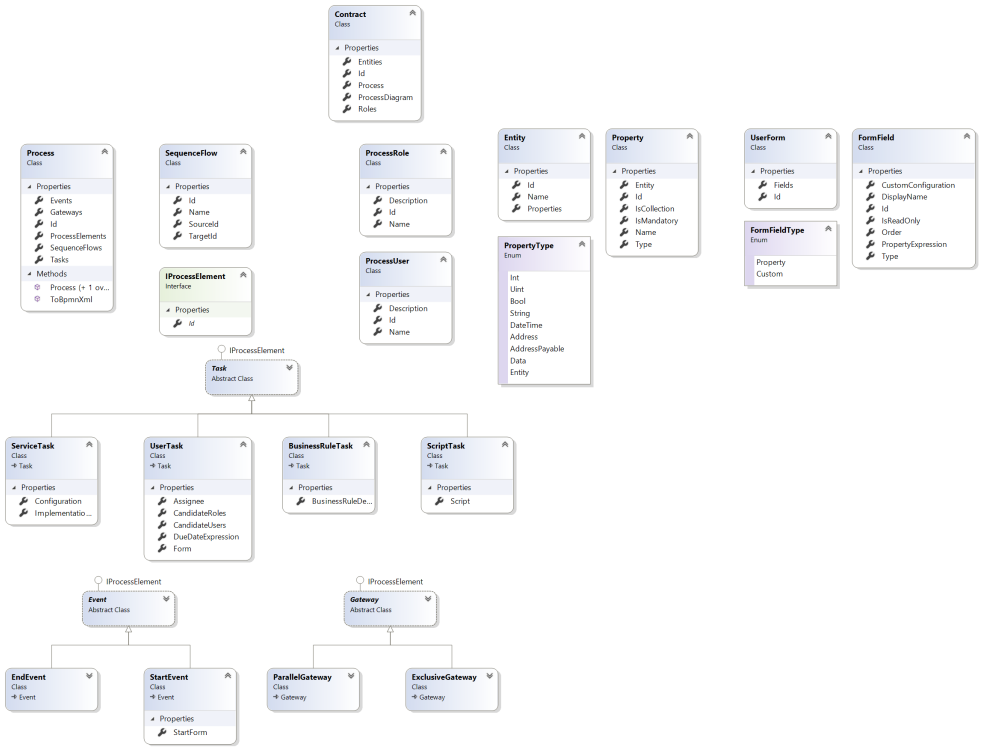

Figure 4.1: Contract Diagram[6]

Afterwards, generating of Solidity code from the `Contract` class begins. New `SolidityContract` class is initialized consistsing of list of `SolidityComponents` that is further appended. Firstly the two mappings mentioned in section 3.2.2.2 are added to the contract outside of algorithm as well as `isStateActive` function. At this time, the data model is also added – in form of structures.

Next, the main algorithm begins. The `IterateProcess` method uses BFS algorithm to go through every element and add it to the `SolidityContract` object, starting from the unique `StartEvent` (Solidity contract constructor).

When this process is completed, output Solidity contract string can be generated by `GenerateSolidity` method. `SolidityContract` and all components are using LiquidString as templates.

4.3 Mortgage Proof-of-Concept

The current state of mortgage acceptance process is described in paper by Ing. Barbora Hornáčková [7] as follows:

The mortgage contract is a complex process involving several parties, dependent processes, level of trust between parties and a lot of documents proving results of auxiliary processes; Notarization is involved for all parts. These aspects all contribute to overall complexity and costs of the process. Thus it appears as a good use case, where modeling by DEMO would capture the essence of the process and a smart contract could offer an automated notarization, data sharing between parties and payment processing, thus reducing the need of manual processes, as illustrated in Figure 4.2.

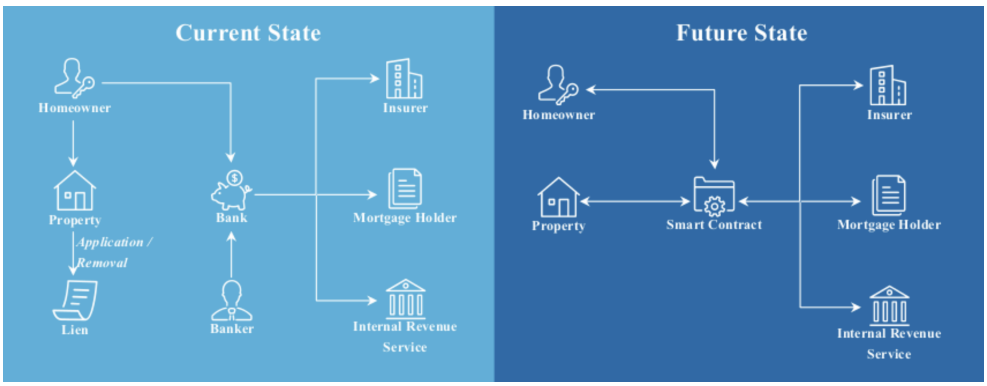

Figure 4.2: Mortgage process changed using smart contract[7]

As stated in the citation, mortgage contracts are complex currently very expensive and time consuming processes while also prone to human mistakes. This is where the modern technologies, specifically blockchain and smart contracts, could help.

Writing the smart contract with all loopholes in mind can be however as difficult as the current process. This is where DasContract helps, providing easy to understand interface of graphical modelling, while the production of smart contract code is taken care of by background processes. This same or slightly changed model can be further reused in multiple similar cases, e.g. in our use case example the same smart contract code can be used for as many mortgage contracts as needed with no changes at all, or only minor ones like indemnity terms or minimal down payment.

4.3.1 Mortgage Proof-of-Concept diagram

For the mentioned mortgage contract use case, the DasContract diagram was designed[6]. The slightly modified of this contract is shown on Figure 4.3. This model, while detailed, is still not in production ready state and should not be used in real world circumstances.

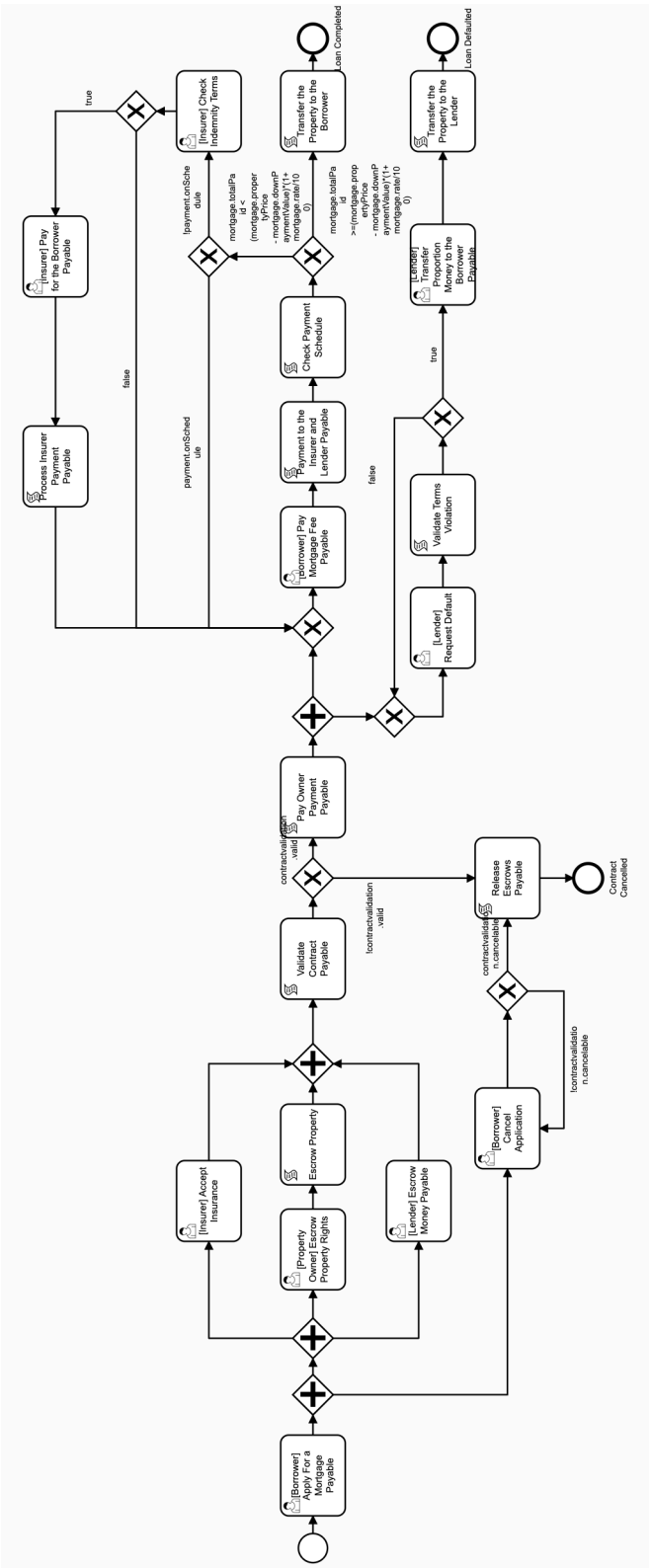

Figure 4.3: Mortgage diagram in DasContract editor[8]

As seen on the diagram, four parties are involved, marked as Borrower, Insurer, Property Owner and Lender. The Borrower starts the contract by transacting the first activity and filling details about the mortgage contract as parameters.

Next, two parallel gateways follow. The first one gives Borrower choice to cancel application up to the point where the contract is validated. The second parallel gateway opens the states for other parties to accept or decline their involvement in the contract with its parameters as filled by Borrower. If one or more parties don't agree to the terms, contract will not validate, ends and returns any escrow money or property tokens to the original owners.

After the contract's successful validation the Property Owner is paid the money for his property and is out of scope for the rest of contract. Then another parallel region begins.

On one side of the parallel region is the regular payment logic, where Borrower pays for the mortgage in monthly payments. Monthly payments are however in this simplified use case replaced by the predefined total value of mortgage and is considered being behind payment schedule if the sum of payments is lower than monthly payment multiplied by number of payments made. If this is the case, Insurer will check whether or not were indemnity terms met. If so, Insurer will pay the required amount for this month instead of Borrower. With each payment, most of the money goes to Lender, while a small percentage is claimed by Insurer. After the mortgage is paid in its entirety, the property is automatically transferred to Borrower.

Second parallel region is for the Lender to use in case he isn't receiving money which means that Borrower is not paying his monthly fee and neither is Insurer because the indemnity terms are not met. In that case Lender will request for Borrower's default and the terms are verified. If the verification process passes, Lender will pay proportion of money (defined in terms) to Borrower and the property is transferred to Lender.

4.3.2 Code example

Both DasContract and Solidity code examples were already explained in Chapter 3. Here we will however show and explain the important logic of our example mortgage use case contract – the generated `Mortgage` structure and Solidity code provided in script activities. The code was generated by program developed in practical part of this thesis and is publicly available under MIT Licence on `https://github.com/CCMiResearch/DasContract` and included as attachment with this thesis.

First we will look at the Solidity `Mortgage` structure code generated from DasContract data model:

```
struct Mortgage{
        uint256 propertyTokenID;
        uint256 propertyPrice;
```

```
        uint256 rate;
        uint256 downPaymentValue;
        uint256 mortgageDurationMonths;
        uint256 totalPaid;
        Escrow escrow;
        mapping (uint => Payment) payments;
        uint paymentsLength;
}
```

`Mortgage` is the base structure of whole contract. It contains information about mortgage and its state, containing all payments in the `payments` mapping and helper variable `paymentsLength` for keeping track of the number of payments since mappings in Solidity don't implement the `length()` function. The first five variables are filled out by borrower in function parameters when applying for mortgage.

Interesting variable in the `Mortgage` structure is `propertyTokenID`. It contains the ID of ERC721 token implementation representing the house ownership. This kind of ownership representation would have to be implemented on national level before the mortgage contract could exist inside blockchain.

Another important part of the example mortgage contracts are script activities. As discussed in section 3.2.3, scripts itself aren't automatically generated and have to be manually created for each script activity. Generated part of smart contract secure the flow, transaction security and to some extent data logic, especially saving parameters of functions to given structures and variables. The script activity part is main logic of contract in this example the function responsible for validating mortgage contract (formatting slightly modified):

```
function ValidateContract() isValidateConState internal
{
    ActiveStates["ValidateContract"] = false;
    if (payable(address(this)).balance
        >=
        mortgage.propertyPrice
        &&
        insurance.isEnsured)
        {
            contractvalidation.valid = true;
            contractvalidation.cancelable = false;
        }
        else
        {
            contractvalidation.valid = false;
        }
```

```
    Gateway_1s86rvzLogic();
}
```

In this example code, the whole function body except first and last line had to be manually written since there currently is no universal model for this kind of deeper, very contract specific, logic implemented in DasContract. The function `ValidateContract()` checks whether all requirements have been met. Specifically if contract has enough money from the lender to pay property owner for the property and also if the borrower is insured. Check for property ownership is defined in previous state `Escrow Property` because of Solidity requirements.

4.4 Testing

For testing, two approaches were selected. For testing code generator itself, unit tests are included as separated project in DasContract .NET Core solution. Second type of testing is manual one using the Remix platform.

4.4.1 Unit testing in .NET Core

The unit tests are testing the component part of project. Mainly the string generated from components are being tested, including structures, functions, modifiers or if-else blocks. There are totally 22 unit tests included testing 8 component classes, all of them passing.

4.4.2 Manual testing using Remix

Testing generated Solidity code itself was made by manual testing and debugging while developing the program. The testing was made in Remix IDE and detailed debugging functions were used.

This isn't very scientific testing and Solidity does have support for unit tests, manual testing is however the better testing solution for this project as the unit tests would be different for each generated contract.

4.5 Chapter summary

In the last chapter, we took a look at used technologies that were used in process of development and described the flow by which is the Solidity code generated.

The problem of current state of mortgage contracts were explained and the possible solution using blockchain technology combined with DasContract designing was introduced. While both blockchain and DasContract still have its flaws, and aren't quite ready yet for real world usage, they represent the concept that may be used for all kinds of contracts in near future.

Further, the flow and activities of said mortgage DasContract example was explained with Solidity code examples of `Mortgage` structure and the `ValidateContract()` script function. The representation of property using ERC721 tokens was also explained.

The last part of this chapter was dedicated to two approaches to testing that were used while programming this project and mortgage contract. It was also explained why in our case the manual testing of generated Solidity code is better than the unit testing.

Conclusion

Thesis demonstrated usage of Ethereum blockchain as decentralized storage and computer for smart contracts generated from DasContract files. This solution is very cost effective with controlled ambiguity by the code in decentralized storage. Using blockchain for these purposes might be possible in the future, however currently it works only on Proof-of-Concept level. No public blockchain is yet on the required level of security, scalability and decentralization to be able to trust them with our property and commitments.

Both DasContract and the Solidity generator implemented as part of this thesis are great starting points, but have long way before being production ready. They need proper testing and bug fixing as well as new features. For DasContract, the ability to include parts of scripts inside the user tasks would be good for diagram clarity as it might reduce the number of script tasks.

More important concept needed that DasContract already assumes, but was not implemented in either DasContract Editor[17] or this thesis are oraclas discussed in section 1.2.3. They could be then used as source of information from outside of blockchain, e.g. in our mortgage example for confirmation of borrower's identity and his liabilities. Another concept that is now being added to DasContract is decentralized identity, which could remove the need of aliases for each and every contract and increase the overall contract security and trust.

For future research, this thesis submits and makes a proposal to increase the amount of generated code by introduction new features to all DasContract, DasContract editor and Solidity converter. One of those features could be payment needed to transact function as payments have proven very often used in mortgage example and while testing Solidity smart contracts. There is also opportunity to generate code from DasContract files to other languages than Solidity, e.g. Vyper or any other new language for Ethereum, Ethereum 2.0 or even for some other blockchain that would be globally recognized by community and hopefully one day by governments.

Bibliography

[1] Tar, A. Proof-of-Work, Explained. January 2018, [cit. 2020-02-14]. Available from: https://cointelegraph.com/explained/proof-of-work-explained

[2] Yeaow, A. Global Bitcoin Nodes Distribution. July 2020, [cit. 2020-07-12]. Available from: https://bitnodes.io

[3] Tam, K. First Attempt on Vyper. January 2019, [cit. 2020-07-15]. Available from: https://medium.com/coinmonks/first-attempt-on-vyper-eb1d1ccea6ed

[4] Song, X. W. Z. Z. D. *Learn Ethereum*. Packt Publishing Ltd., first edition, ISBN 978-1-78995-411-1.

[5] Skotnica, M.; Pergl, R. *Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts*. 01 2020, ISBN 978-3-030-37932-2, pp. 149–166, doi:10.1007/978-3-030-37933-9_10.

[6] Skotnica, M. CCMiResearch/DasContract on Github [online]. [cit. 2020-07-29]. Available from: https://github.com/CCMiResearch/DasContract/

[7] Hornáčková, B.; Skotnica, M.; et al. Exploring a role of blockchain smart contracts in enterprise engineering. In *Enterprise Engineering Working Conference*, Springer, 2018, pp. 113–127.

[8] Skotnica, M. CCMiResearch/DasContract on Github [online]. [cit. 2020-07-30]. Available from: https://github.com/CCMiResearch/DEMOCaseStudies/tree/master/Blockchain/Mortgage

[9] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008, [cit. 2020-02-14]. Available from: https://www.bitcoin.com/bitcoin.pdf

[10] Lipovyanov, P. *Blockchain for Business 2019*. Packt Publishing Ltd., first edition, ISBN 978-1-78995-602-3.

[11] Rosic, A. What is Cardano Blockchain? 2018, [cit. 2020-07-13]. Available from: https://blockgeeks.com/guides/what-is-cardano/

[12] Dale, B. Everyone's Worst Fears About EOS Are Proving True. September 2019, [cit. 2020-07-13]. Available from: https://www.coindesk.com/everyones-worst-fears-about-eos-are-proving-true

[13] A complete list of VeChain partnerships. July 2020, [cit. 2020-07-15]. Available from: https://vechaininsider.com/partnerships/a-complete-list-of-vechain-partnerships/

[14] Wood, A. M. A. D. G. *Mastering Ethereum*. O'Reilly Media, Inc., first edition, ISBN 978-1-491-97194-9.

[15] Frumkin, D. What Are Ethereum Tokens? ERC-20, ERC-223, ERC-721, And ERC-777 Tokens Explained. July 2018, [cit. 2020-07-13]. Available from: https://www.investinblockchain.com/what-are-ethereum-tokens/

[16] Dixit, K. Ethereum 2.0 Wants To Save Its Predecessor Its Blushes; Here's How Its Planning To Do It. January 2020, [cit. 2020-02-15]. Available from: https://www.tronweekly.com/news-eth-ethereum-2-0-wants-to-save-its-predecessor-its-blushes-heres-how-its-planning-to-do-it/

[17] Drozdík, M. Open-Source Legal Process Designer in .NET Blazor. May 2020. Available from: https://dspace.cvut.cz/bitstream/handle/10467/88271/F8-BP-2020-Drozdik-Martin-thesis.pdf

[18] What is BPMN? [online]. [cit. 2020-07-15]. Available from: https://www.visual-paradigm.com/guide/bpmn/what-is-bpmn

[19] López-Pintado, O.; García-Bañuelos, L.; et al. Caterpillar: A business process execution engine on the Ethereum blockchain. *Software: Practice and Experience*, May 2019, ISSN 1097-024X, doi:10.1002/spe.2702. Available from: http://dx.doi.org/10.1002/spe.2702

[20] Cummings, S. The Four Blockchain Generations. February 2019, [cit. 2020-07-22]. Available from: https://medium.com/the-capital/the-four-blockchain-generations-5627ef666f3b

[21] Partz, H. VeChain Loses $6.6M in VET Tokens to Hacker in Attack on Buyback Wallet. December 2019, [cit. 2020-07-22]. Available from: https://cointelegraph.com/news/vechain-loses-11b-vet-tokens-to-hacker-in-attack-on-buyback-wallet

[22] PoW 51% Attack Cost [online]. [cit. 2020-07-23]. Available from: https://www.crypto51.app/

[23] Ethereum Nodes Map [online]. [cit. 2020-07-23]. Available from: https://matallo.carto.com/builder/e70677d5-1111-40a8-9e19-f27da227a55c/embed

[24] Pirus, B. ETH Scalability Isn't Going to Be an Issue Soon, Suggests Vitalik Buterin. June 2020, [cit. 2020-07-23]. Available from: https://cointelegraph.com/news/eth-scalability-isn-t-going-to-be-an-issue-soon-vitalik-buterin-posits

[25] ETH Gas Station [online]. [cit. 2020-07-23]. Available from: https://ethgasstation.info/

[26] Remix - Ethereum IDE [online]. [cit. 2020-07-30]. Available from: https://remix-ide.readthedocs.io/en/latest/index.html

[27] Solidity Documentation [online]. [cit. 2020-07-25]. Available from: https://solidity.readthedocs.io/

Appendix A

Acronyms

XML Extensible markup language

BMPN Business Process Model and Notation

PoS Proof-of-Stake

PoW Proof-of-Work

PoA Proof-of-Authority

USD United States dollar

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `DasContract` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format