ASSIGNMENT OF BACHELOR’S THESIS

Title:	Generating Plutus Smart Contracts from DEMO Process Models
Student:	Ondřej Šelder
Supervisor:	Ing. Marek Skotnica
Study Programme:	Informatics
Study Branch:	Information Systems and Management
Department:	Department of Software Engineering
Validity:	Until the end of winter 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 Plutus that is hard to understand by humans and is prone to errors. Based on preliminary research, DEMO models seem to provide a better way to define smart contracts. A goal of this thesis is to propose a way how to generate Plutus smart contracts from DEMO models.

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

References

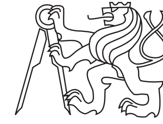
Will be provided by the supervisor.

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

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

Prague May 26, 2019

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering



Bachelor's thesis

Generating Plutus Smart Contracts from DEMO Process Models

Ondřej Šelder

Supervisor: Ing. Marek Skotnica

7th January 2020

Acknowledgements

I would like to express my gratitude to everyone who helped me or supported me to complete my thesis. First and foremost, I would like to thank my supervisor Ing. Marek Skotnica for his help, time and guidance.

My appreciation goes also towards my partner, family, and friends for all their support, not only during the time I was working on this thesis but also during the whole university studies.

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 in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on 7th January 2020 ....

Czech Technical University in Prague
Faculty of Information Technology
© 2020 Ondřej Šelder. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Šelder, Ondřej. *Generating Plutus Smart Contracts from DEMO Process Models*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

The current way of people conducting contracts can be updated in the future using blockchain technology. This decentralized automated system supports the concept of smart contracts, codes that extend the possibilities of the blockchain. The Plutus programming language is a domain-specific language for writing smart contracts. The disadvantage of this language is that it is hard to understand by people without advanced skills. Defining smart contracts with DEMO methodology models solves this obstacle and even reduces error-prone. The main goal of this thesis is to propose a way how to generate Plutus smart contracts from DEMO process models.

This thesis evaluates the benefits of the Cardano project, which is developing the Plutus platform for writing smart contracts. Furthermore, it introduces an approach of the generation using templates and a concept of state machines. The last part of the text demonstrates this generative process on an actual use-case of land title transfer recording. An implementation of an algorithm that performs the process of a model conversion to a smart contract can also be found in the attachment.

Keywords blockchain, smart contract, Cardano, Plutus, DEMO methodology, process models

Abstrakt

Technologie blockchain v budoucnu změní způsob, jakým dnes lidé uzavírají smlouvy. Tento decentralizovaný automatizovaný systém podporuje koncept smart kontraktů, kódů, které rozšiřují možnosti blockchainu. Jedním z programovacích jazyků, které slouží pro psaní smart kontraktů, je doménově specifický jazyk Plutus. Nevýhodou tohoto jazyka je jeho náročné pochopení lidmi bez pokročilých znalostí. Definování smart kontraktů pomocí modelů metodiky DEMO tuto překážku odstraňuje, a dokonce snižuje náchylnost k chybám. Hlavním cílem této práce je navrhnout způsob, jak generovat smart kontrakty Plutus z procesních modelů DEMO.

Tato práce hodnotí přínosy projektu Cardano, který vyvíjí platformu Plutus pro psaní smart kontraktů. Dále představuje způsob generace pomocí šablon a konceptu stavových automatů. Poslední část textu ukazuje tento generativní proces na skutečném případu užití, který se týká převodu vlastnictví pozemku. V příloze je také implementace algoritmu, který provádí proces překladu modelů DEMO na smart kontrakt.

Klíčová slova blockchain, smart kontrakt, Cardano, Plutus, DEMO metodika, procesní modely

Contents

Introduction **1**

1 Theoretical Foundations **3**
1.1 Blockchain Technology 3
1.2 Smart Contracts and Blockchain 2.0 7
1.3 Cardano Blockchain 9
1.4 Business Process Modeling (BPM) 13
1.5 Chapter Summary 18

2 Plutus Smart Contract Generation from DEMO Models **19**
2.1 DEMO and Extended UTXO Compatibility 19
2.2 Generation of On-Chain Code 23
2.3 Generation of Off-Chain Code 30
2.4 Chapter Summary 35

3 Proof of Concept **37**
3.1 Used Technologies 37
3.2 Software Architecture 38
3.3 Testing 39
3.4 Chapter Summary 47

Conclusion **49**

Bibliography **51**

A Acronyms **55**

B Contents of enclosed CD **57**

List of Figures

1.1 Graphic Representation of the Blockchain 4
1.2 Unspent Transaction Output Definitions 5
1.3 Plutus Tx Code Example 11
1.4 Plutus Core Code Example 11
1.5 Plutus Tx Function Example 12
1.6 The Four Ontological Submodels 14
1.7 Transaction Pattern Diagram 16

2.1 Contract Meta-model: UML Diagram 21
2.2 Transaction Axiom State Machine 22
2.3 Generated Pseudo-code: Process ID 24
2.4 Generated Pseudo-code: Data Type Representing Facts 25
2.5 Generated Pseudo-code: Initiator and Executor Actor Roles 25
2.6 Generated Pseudo-code: States of the Transaction Kinds' State Machines 26
2.7 Generated Pseudo-code: Inputs of the Transaction Kinds' State Machines 27
2.8 Generated Pseudo-code: Transition Functions of the Transaction Kinds' State Machines 28
2.9 Generated Pseudo-code: Check Functions of the State Machines . 29
2.10 Generated Pseudo-code: Final Functions of the State Machines . . 29
2.11 Generated Pseudo-code: Transaction Kinds' State Machines 30
2.12 Generated Pseudo-code: Transaction Kinds' State Machines Instance and Client 31
2.13 Generated Pseudo-code: Parameters of the Endpoint Functions . . 32
2.14 Generated Pseudo-code: Schema Definition 32
2.15 Generated Pseudo-code: Endpoint Functions 33
2.16 Generated Pseudo-code: Endpoint Functions 34
2.17 Generated Pseudo-code: Registration of the Endpoints 34

3.1 Component Diagram of the Implementation 39
3.2 Land Title Transfer Recording: Processes' Approaches Compared . 41
3.3 Land Title Transfer Recording: OCD 42
3.4 Land Title Transfer Recording: PSD 43
3.5 Land Title Transfer Recording: OFD 44
3.6 Land Title Transfer Recording: Action Rule Resolving Transfer . . 44
3.7 Simulation: Initializing the Land Title Transfer Recording 45
3.8 Simulation: Sending the Land Title Transfer 45
3.9 Simulation: Sending the Payment 46
3.10 Simulation: Resolving the Land Title Transfer Recording 46

Introduction

Since 2003, the time required to enforce a contract (the day from filing the lawsuit until its closure) has increased and normally exceeds 600 days. [1] Because of that, a well-designed contract is a crucial part of all legal agreements. For its completion, it is necessary to deeply understand contract law which deals with detailed requirements for both sides of the deal. Because these subjects must abide by so much information, there is a need, and in many cases, it is obligatory, to hire a third party to oversee the transaction.

The contract comes to importance when one party to the contract breaches the terms of the contract. In that case, the legal system has to intervene. In the process of legal fact-finding related to the case, a great amount of money and time is invested by both sides.

Using information technology approaches it is possible to save costs in these situations and eliminate many unnecessary middle parties. The blockchain network has been proven to be secure and suitable for storing and managing money transfers. A recent generation of blockchain with suffix 2.0 allows extended functionality of the network via executable code. This feature of so-called smart contracts is applicable in the use-case mentioned above and also in many different fields but mostly in finance, law and state administration.

Cardano is an open-source blockchain platform for smart contracts. Its developing company Input Output HK Limited marks it as a third-generation blockchain. It aims primarily at solutions brought by not only their developers but also by academia, and business. For the construction of smart contracts the programming language Plutus is used. Unlike imperative languages, it is built as a Haskell-like functional programming language.

A possible disadvantage in the future may be regarding the implementation of such a blockchain solution by professionals with little knowledge in computer science. This problem is solved by communicating the concepts to programmers familiar with blockchain and smart contract developing. That still leads to expensive and ineffective work and it increases the risk of causing a fault that would be permanent in the blockchain network.

The Design and Engineering Methodology for Organizations (DEMO) is a modeling methodology used in business. It not only models enterprise processes but has also been proven to be suitable for describing a smart contract. [2] Generating a smart contract from this concept model saves money, time and lowers the risk of error and serves as a useful tool to non-programmers who need to implement a blockchain network.

The goals of this thesis are to explore the state-of-the-art of the Cardano blockchain and to propose a way to generate Plutus smart contracts from DEMO models. The thesis will also analyze and implement a smart contract generator for one specific programming language.

Structure of the Thesis

The thesis is structured as follows:

Chapter 1 – Theoretical Foundations – provides information about blockchain and smart contracts to understand basics which are then subsequently used. It describes the state-of-the-art Cardano blockchain technology and its strengths and weaknesses in contrast to other approaches. Lastly, the section outlines DEMO Methodology and its models.

Chapter 2 – Plutus Smart Contract Generation from DEMO Models – contemplates a suitable structure to record DEMO models and finds ways to interpret it as a Plutus smart contract.

Chapter 3 – Proof of Concept – describes own implementation of Plutus smart contract generator: the technologies used and a draft of the software architecture. This chapter also introduces one possible use-case of blockchain technology, converts it into the DEMO model and uses it as an example for simulation.

Conclusion – sums up and evaluates the used solution to generate a Plutus smart contract from the DEMO process models, ensures that research goals have been met, and offers possible directions for future research and implementation.

Chapter 1

Theoretical Foundations

In this chapter, the thesis will lay background information an concepts that are needed to generate Plutus smart contracts from process models of the DEMO methodology. The chapter begins with describing blockchain and its cores to understand the advantages and limitations of the technology to support the goals and motivation behind the generation itself. It also leads to a definition of what a smart contract is. Then the programming language Plutus of Cardano blockchain is explored as it was chosen for writing smart contracts in the thesis. Finally, the DEMO methodology and its models are outlined.

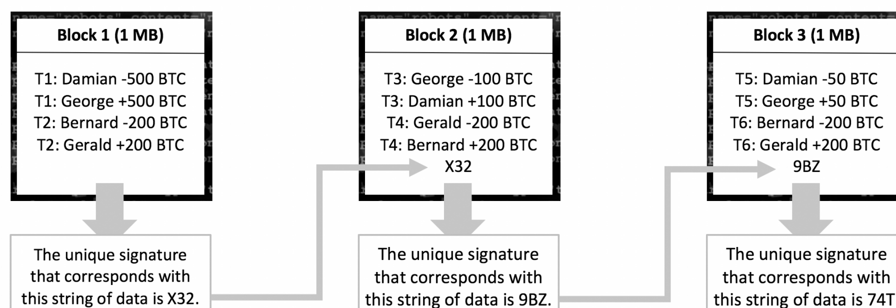
1.1 Blockchain Technology

Fundamentally, the blockchain network is a set of nodes that are trying to achieve state replication. [3] This technology combines many others introduced decades [3] before the release of the first conceptualization that was Bitcoin cryptocurrency in 2008 by Satoshi Nakamoto. [4] The concept of Bitcoin and blockchain in general works on four essential principles: Decentralization, Transparency, Immutability, and Security. These principles are achieved by following information technology approaches: Distributed ledger technology (DLT), cryptography, consensus mechanism, peer-to-peer network, and software code base. [5]

1.1.1 Distributed Ledger Technology

Blockchain database filesystem contains specifically stored data in the form of lists of records. Otherwise, the terms chains of blocks or digital ledgers are used. Data are built up and stored in successive blocks [7]. Each of these blocks includes data, timestamp, digital signature and additional hash that refers to the previous block. [2] This ensures that any manipulation with records in the blockchain can be detected.

Figure 1.1: Graphic Representation of the Blockchain[6]



All the users of blockchain have control over the data validity because each one of them is storing an identical copy of the database. If any participator wants to perform a new transaction, it needs to be verified by some portion of other participants in the network. After the verification, a new block is created and connected on top of the last block. [5]

This process is irreversible and the record is permanent. All mentioned delivers security that is otherwise entrusted to middle-party despite created redundancy, which is actually the tool of distributed systems to tolerate node failures. [3] Exemplary graphic representation of the blockchain's core idea can be seen in Figure 1.1.

1.1.2 Cryptographic Techniques in Blockchain

"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." [4]

Blockchain has to use many techniques of cryptography in different parts of its filesystem. For example, Bitcoin is dependent on the use of public and private keys, hash functions and digital signatures. [5]

The user's node is represented by a public address that is derived from the public key. The transaction is sent to the address and the virtual money is accessible only with the private key of the account's owner. This also excludes any restoration of access when the private key is lost and responsibility is entirely on the user. [5]

Block can hold a certain amount of transactions in the size of Bytes. If the user stops sending transactions or he reaches the limit, the previous block is calculated into a hash that is then stored in the new block of transactions. [2] Hash functions are also used in the Proof of Work consensus algorithm. Each

block is also marked with a digital signature created through an algorithm from private key and transaction. [5]

1.1.3 Unspent Transaction Output

Traditional accounting views the flow of money as transfers between two different bank accounts. Bitcoin and other platforms are using an accounting style called Unspent Transaction Output (UTXO). This approach documents money transfers from one transaction to another. [8]

Figure 1.2: Unspent Transaction Output Definitions[8]

Primitive types

$$\begin{aligned} txid &\in \mathsf{TxId} && \text{transaction id} \\ ix &\in \mathsf{Ix} && \text{index} \\ addr &\in \mathsf{Addr} && \text{address} \\ c &\in \mathsf{Coin} && \text{currency value} \end{aligned}$$

Derived types

$$\begin{aligned} tx &\in \mathsf{Tx} &&= (inputs, outputs) \in \mathbb{P}(\mathsf{TxIn}) \times (\mathsf{Ix} \mapsto \mathsf{TxOut}) && \text{transaction} \\ txin &\in \mathsf{TxIn} &&= (txid, ix) \in \mathsf{TxId} \times \mathsf{Ix} && \text{transaction input} \\ txout &\in \mathsf{TxOut} &&= (addr, c) \in \mathsf{Addr} \times \mathsf{Coin} && \text{transaction output} \\ utxo &\in \mathsf{UTxO} &&= txin \mapsto txout \in \mathsf{TxIn} \mapsto \mathsf{TxOut} && \text{unspent transaction outputs} \\ b &\in \mathsf{Block} &&= tx \in \mathbb{P}(\mathsf{Tx}) && \text{block} \\ pending &\in \mathsf{Pending} &&= tx \in \mathbb{P}(\mathsf{Tx}) && \text{pending transactions} \end{aligned}$$

Functions

$$\begin{aligned} \mathsf{txid} &\in \mathsf{Tx} \to \mathsf{TxId} && \text{compute transaction id} \\ \mathsf{ours} &\in \mathsf{Addr} \to \mathbb{B} && \text{addresses that belong to the wallet} \end{aligned}$$

Filtered sets

$$\begin{aligned} \mathsf{Addr}_{\mathsf{ours}} &= \{a \mid a \in \mathsf{Addr},\ \mathsf{ours}\ a\} \\ \mathsf{TxOut}_{\mathsf{ours}} &= \mathsf{Addr}_{\mathsf{ours}} \times \mathsf{Coin} \end{aligned}$$

It can be seen from the Figure 1.2 that the structure (derived type utxo) that contains the information about the transactions is a finite map with the pair of transaction ID and an index as a key and the pair of an address and a currency value as a map's value. The index is a unique identifier within a set of outputs and the address is referring to the public key of the coin's owner. [8]

The main distinction of UTXO accounting from traditional accounting is that the money being spent comes from unspent transaction outputs (identified by mentioned index) which reside on the blockchain after previous transactions. Aside from the mentioned types, transactions obtain certificates, hashes and other data. [8]

1.1.4 Proof of Work Consensus Algorithm

Decentralized systems must deal with some problems due to the malicious behavior of certain nodes in communication between nodes. One of them was described in a 1982 paper The Byzantine Generals' Problem. [9] The problem

works with the concept of an army consisting of units that need to coordinate an attack with a twist that one of the generals is a traitor. [5]

Proof of Work is one of the implementations of the Byzantine fault tolerance solution. This implementation was introduced in 1999 [10] and Satoshi Nakomoto used it in his Bitcoin blockchain. In this case, the role of a general is represented by a node in the network. The user is called miner and performs mining. Mining is a competitive act, in which miners puts their computational resources to solve Proof of Work mathematical puzzle (finding a hash of the block with a certain amount of zeros at the beginning). The miner who finishes first adds the new block to the network and gets a reward in the form of Bitcoins. [5]

In theory, this solution can be still bypassed but at the cost of too much electric energy consumption. A potential attacker would have to has 51% of all computational power in the blockchain network. And also he would only be able to change blocks in his list of records for example to double-spend money. Other accounts would be secure because the attacker still doesn't have users' private key. [5]

1.1.5 Peer-to-Peer network

The verification process can be time-consuming when the blockchain grows in transactions and therefore in storage size. Because of that, Bitcoin chose an approach in which there are two types of nodes: Full nodes and lightweight nodes. [5]

The difference between them is that full nodes store a full copy of the blockchain, unlike lightweight nodes which store only partial copy. Lightweight nodes or Simplified Payment Verification (SPV) are using a top-down approach from the newest transaction to older parts of the list. It also stores only block headers thus reducing the size. They are used in mobile devices and serves for verification. [5]

Two types of the Bitcoin blockchain (and many others) are distinguished: The mainnet and the testnet. The mainnet is the network as the thesis described it before. The testnet is used for research and testing new Bitcoin protocol features without disruption of the mainnet. [5]

The network is also divided into two types: Public and private blockchain. Although the latter is not recognized by part of the community as a true blockchain. Public blockchain allows anyone with the right hardware to join the network. More users make consensus algorithms more efficient. On the other hand, private blockchains secure the privacy of their operation from unwanted parties by granting permission to the network from the so-called gatekeeper. [5] Often, the terms permissionless and permissioned system is used as an alternative. [3]

1.1.6 Code Base of Bitcoin

The initial protocol was introduced by Satoshi Nakamoto in 2008. [4] After the launch, Satoshi passed the development and maintenance of the Bitcoin to software developers that made it an open-source project. [11]

Bitcoin was constructed as a programmable cryptocurrency. For that purpose serves the high-level programming language Script that was designed limited in scope to run on most of the devices and to reduce coding errors. It allows time-locks, conditional clauses, and functions. Despite that, Script is not a Turing-complete language and that limits it in solving various problems. [5]

Aside from the network, many supporting tools were developed and are available on the platform of the blockchain. Wallets are the basic tool that gives the user access to the blockchain. They are storing users' private keys, therefore, allowing them to access their cryptocurrencies. They could be in the form of desktop wallets, mobile wallets, and web wallets. To increase safety, the offline wallets were created as a hardware wallet or simple paper wallet. [5]

1.2 Smart Contracts and Blockchain 2.0

Previously mentioned code base of Bitcoin and early blockchains were inadequate and restrictive towards this technology full potential. The first generation of blockchain is a designation of all cryptocurrencies based on this technology. Etherum blockchain was the first that allowed programmable blockchain and is marked as Blockchain 2.0. The second-generation blockchains are enhanced by smart contracts and they are used beyond cryptocurrency in financial services. [12]

1.2.1 Smart Contract

"A set of promises, including protocols within which the parties perform on the other promises. The protocols are usually implemented with programs on a computer network, or in other forms of digital electronics, thus these contracts are "smarter" than their paper-based ancestors. No use of artificial intelligence is implied." [13]

The first use of the term smart contract was used and described by Nick Szabo in 1993 [14]. It is an executable code integrated into the blockchain that is executed every time there are certain conditions met. Smart contracts are not necessarily concept dependent on blockchain technology but use it for its security benefits. [12]

Smart contracts are secure and unstoppable thus their implementation has to be thorough and fault-tolerant. They are also not necessarily automatic.

Automatable smart contracts allow manual human input in certain scenarios via Oracles. [12]

Smart contracts need to be deterministic because if any node was provided with different output then consensus in the network is broken. As examples could serve numbers with a floating-point that differs on different hardware. [12]

1.2.2 Smart Contract Templates

It is a question on how to approach smart contract in a court of law that now does not understand code as a contract. It could be more acceptable in legal situations by making it readable both by machines and humans. [12]

Clack et al proposed an idea in their paper *Smart Contract Templates: Foundations, design landscape and research directions* [15] to build standard templates to provide a framework. This framework should support legal agreements for financial instruments.

There has been also research for developing domain-specific languages. These languages are limited in expressiveness to serve a particular application or area of interest. This idea could be further extended to graphic domain-specific languages. A platform working with this concept allows non-programmer domain expert to design financial contract in the graphic interface. [12]

1.2.3 Oracle

The execution of the business logic often requires decision making based on some external data. These data could be provided to the blockchain by Oracles. Oracle is an interface that delivers data from an external source to a smart contract. This could also serve for communication in the Internet of Things (IoT). [12]

Smart contract subscribes to the Oracle and is able to pull the data. Oracles can also push the data to the smart contract. It has to be secured that the data are unchanged in the Oracle and that the data source is reliable and authorized. This backward causes the need for a trusted party that delivers the external data but it is sometimes necessary for certain processes. The problem of trust could be partially solved by making the third party decentralized for example making it private blockchain. [12]

1.2.4 Proof of Stake Consensus Algorithm

The Proof of Work consensus algorithm is becoming ineffective when computing power in the blockchain network is growing. Electricity usage in Bitcoin exceeds that of some countries electricity consumption just for verification. [16] Peercoin first proposed an alternative to this consensus algorithm with Proof of Stake. [12]

The idea of the algorithm is based on the assumption that the user puts into the blockchain enough stake that will discourage him from making any malicious behavior. Mining is here easier for the user who demonstrably owns more coin thus having more stake in correctly verifying the transaction. Again, this consensus could be broken by node owning more than half of all coins but this approach would damage user himself. [12]

1.2.5 Other Consensus Algorithms

Aside from Proof of Work and Proof of Stake algorithms there are other that can manage to gain consensus. In Proof of Stake, all users who have a stake in the blockchain are called stakeholders. In its modified version Delegated Proof of Stake these stakeholders votes to delegate the validation of a transaction. [12]

As an alternative, Intel Corporation introduced Proof of Elapsed Time that uses Trusted Execution Environment (TEE) to provide randomness and safety in the leader election process via a guaranteed wait time. The disadvantage of this consensus algorithm is that it requires only Intel processors with Software Guard Extension (SGX). [12]

Proof of Importance extends Proof of Stake with consideration of preferring user who is more active in the network. Proof of Space or Proof of Capacity is a similar concept to PoW except for using disk storage space instead of computing resources of the device. There are many other implementations of consensus. For example Practical Byzantine Fault Tolerant, Tangle, Deposit-based consensus, etc. [17]

1.3 Cardano Blockchain

Cardano is a blockchain platform developed by a company Input Output HK Limited (IOHK). This platform patronizes various software around Cardano Blockchain Mainnet. [18]

The company was created in 2015 when one of its creators Charles Hoskinson did not agree with Etherum founder Vitalik Buterin about future direction. They split up and Hoskinson brought together a team of scientists and engineers to start developing Cardano Blockchain. [19] Aside from Charles Hoskinson, Jeremy Wood is stated as the founder. [18]

1.3.1 ADA and Ouroboros Algorithm

ADA is a cryptocurrency of the Cardano network. It is used for the movement of digital funds but it also serves as a cryptocurrency for internal transactions in the smart contracts. [20]

It features the Ouroboros algorithm which is a variation of Proof of Stake algorithm. IOHK marks other description Proofs of Stake algorithms as super-

ficial and dealing with only some types of attacks. They introduced a rigorous Proof of Stake algorithm that is promoted as a first provable algorithm of its kind. [21]

From a pool of stakeholders, the slot leader is randomly chosen and he is allowed to mine. He is elected in a process called fair lottery by electors, stakeholders who have a certain amount of stake. For creating the randomness the multiparty computation (MPC) is used. The Ouroboros protocol split time into epochs and then to shorter periods of time slots. For each slot, the slot leader is assigned and he has the right to mine only one block during this slot. The election of the slot leader is divided into four phases: Commitment phase, Reveal phase, Recovery phase, and Follow the Satoshi. [21]

1.3.2 Extended Unspent Transaction Output

The basic UTXO model is limited strictly to digital currency accounting. The extended UTXO is a combination and an extension of the UTXO-based model and Etherum's account-based scripting model. Its two main components are an extension to the data carried by the transaction and an extension to the wallet backend. [8]

The first component consists of the validation of the transaction and witnessing to every action the transaction is performing. The scripts are a way to add functionality to the smart contract and are carried by the transaction. These expressions have their own address that can hold coins until certain conditions and processing. [8]

The main script is a validator function stored in a map with the corresponding hash. It returns a boolean value representing the validation result. It is a function of three types: data script, redeemer script, and transaction and ledger data. The data script is carried by a transaction that is paying to the validator script and the redeemer script is carried by a transaction that is collecting from the validation script. The data scripts are stored full on the digital ledger. It also should be noted, that only on-chain code is compiled into the Plutus Core. [8]

1.3.3 Plutus Platform

Authors of an ebook about Plutus programming language Brünjes and Vinogradova [8] describe Plutus as *"a functional development and execution platform for distributed contract applications on the Cardano settlement and computational layers."* In Plutus' software development kit, the libraries' APIs, tools and documentation are available. [22] One of them, Plutus Playground, is an environment for writing, compiling and simulating smart contracts on the blockchain without setting up a full blockchain despite using same programming language and library interfaces as the smart contracts deployed to the Cardano mainnet. It is also available online. [23]

1.3.4 Plutus Core and Plutus Tx

Several components of Plutus Platform are called Plutus or its variations. Plutus Core is the assembly programming language for running smart contracts on the Cardano blockchain. The Plutus Tx is the actual language for writing smart contracts. [8]

Figure 1.3: Plutus Tx Code Example[8]

```
1  integerIdentity :: CompiledCode (Integer -> Integer)
2  integerIdentity = $$(compile [|| \(x:: Integer) -> x
       ||])
```

Figure 1.4: Plutus Core Code Example[8]

```
1  {- |
2  >>> pretty $ getPlc integerIdentity
3  (program 1.0.0
4      (lam ds [(con integer) (con 8)] ds)
5  )
6  -}
```

To highlight the distinction between these programming languages the code examples are provided in this section. The Plutus Tx code is shown in Figure 1.3 that is more clear to the human and Figure 1.4 shows the same code translated into the Plutus Core that is actually running on the blockchain.

Plutus was developed as a functional programming language based on Haskell. It utilizes the safety advantages of Haskell that are necessary to run on distributed systems. Plutus was also designed to be easier to analyze than Haskell. One of Plutus' most prominent features is its automatic separation of code executed by the wallet (off-chain code) and the internal code of the network (on-chain code) that could be written both in the same smart contract. It also should be noted, that only on-chain is compiled into the Plutus Core. [8]

1.3.5 Functional Principles of Haskell

Haskell is a programming language created in the late 1980s and extended in 2010. It is characterized by three main attributes: purely functional, lazy and statically typed. [24]

The first attribute refers to the functional programming paradigm, in which a programmer looks at computation as on the evaluation of mathematical functions. [25] This specific style for solving problems is supported by features of Haskell and its design restrictions make it purely functional. That

means that all variables (in Haskell treated as functions) are immutable and cannot be changed. All adjustments are done by creating a new processed value from the original. [24]

The laziness manages that a specific part of code is never executed until it is needed in some computation. This attribute adds to programs' performance a clearness of the algorithms. It also allows having concepts like infinite arrays. [24]

Compared to languages Perl, Javascript or Python that are dynamically typed, Haskell ranks among programming languages like Java and C# that are statically typed. Meaning that compilers guarantees for strict usage of types only in the place where it can be used. [24]

1.3.6 Constructs of Plutus Tx

Because the Plutus Tx is heavily based on the Haskell functional programming language, the syntax is also very similar. An example of the function's definition (in this case incrementing an integer by one) without compilation can be seen in Figure 1.5.

Figure 1.5: Plutus Tx Function Example[8]

```
1  plusOne :: Integer -> Integer
2  plusOne x = x `addInteger` 1
```

Most of the constructs in Haskell and Plutus behaves like a function. Variable is just a function returning its value and that applies for arrays and more complex data structures too. This raises limitation, that user cannot have two identical names of members in two different data types because getting the member is just function named as the member with its data type as a parameter. [24]

There is also an exception when defining pieces of the code dynamically meaning running the main program. It can be done by writing the static code (being a function) and then pass it as an argument at runtime by so-called lifting. [8]

1.3.7 Haskell's Monads in Plutus

The concept of monads in Haskell is one of its main and powerful features. Because states of objects in Haskell cannot change as easily as in imperative programming, the communication with input and output devices must be designed differently to preserve the functional paradigm. The monad used for this example is monadic type IO. Monad is basically a container where the packed content cannot be unpacked. [26]

Monadic functions are functions that have a monad as an output value type. Every monad supports return function (not the same concept as in imperative programming) which packages an ordinary value inside the monadic function. Bind function is another essential function that allows us to make operations on monads by unpacking, computing and packing them again. For making the code more readable and intuitive, the do-notation was implemented inside monadic functions. [26]

One of the big advantages of the Plutus Tx is that the programmer can write on-chain code together with off-chain code. Off-chain code is represented by the monadic computation of type class MonadWallet or any monad which implements it. [8]

1.3.8 Advantages and Disadvantages of Cardano

The team that is developing Cardano blockchain has chosen a very specific methodology that is the main advantage of the platform that differs itself from other projects. Cardano methodology is based on the academic research, prototyping, technical specifications and rigorous formal development methods. At last, the functional paradigm approach is very unambiguous and less prone to human errors. Functional programming is also easy to mathematically verify and test. [27]

The limitations that come with the mentioned approach could be seen as disadvantages but all decisions are made by the nature of the technology. On the other hand, the Plutus platform is missing many functionalities in the time of writing this thesis that is planned for the future. [27]

1.4 Business Process Modeling (BPM)

For the constant analysis and improvement in the processes of the enterprise the Business Process Management lifecycle was introduced. Aside from its simulation instruments, it is formed by Business Process Modeling (BPM). In general, the model is a description and abstraction of a system. Its conceptual model represents entities and relations between them. A process model is a conceptualization of the business processes in an enterprise. These types of models are often considered synonyms. [28]

1.4.1 Purpose of Business Process Modeling

One of the reasons for BPM is in relation to modeling and examining all the aspects of the enterprise (Enterprise Engineering). BPM is crucial for analysis, design, engineering, and implementation of enterprises because the first step in information system development's successful implementation is to understand the business processes of an organization. [28]

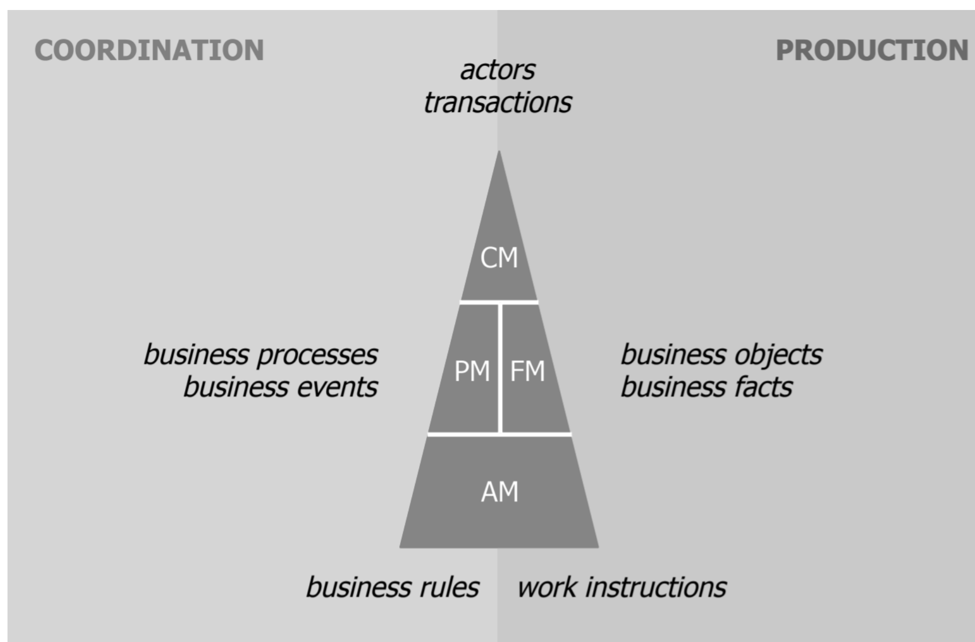
Phalp [29] was describing three elements that came out of his observation and together they form purposes of the process modeling: capturing, analyzing and presenting the business process. Capturing serves for the business users to save time in understanding complex and formal approaches. Analyzing allows the modeler to gain a thorough understanding of the processes, also significantly on account of executable models. The need to easily understand the business processes is met by presenting the purpose of the process modeling. All these aspects improve efficiency and decrease the number of errors in the processes in an enterprise. [28]

1.4.2 DEMO Methodology

When modeling processes, the methodology is a set of methods, rules, and notations. DEMO [30] is a methodology that aims at modeling the construction and the operation of an organization. [28] It was first described by Dietz [31] and then developed and improved by the academic community and it has a deep theoretical background. The main developing purpose of DEMO Methodology is for the creation of an ontological model of an enterprise. [28]

It is based on Performance in Social Interaction (PSI) theory and it is consisting (as seen in Figure 1.6) of four types of models: Construction Model, Process Model, Action Model, and Fact Model. [28]

Figure 1.6: The Four Ontological Submodels[32]



Throughout the time, the DEMO methodology has been created developing its extensive theory, ways to represent DEMO process models in the software language were explored [33] and a big amount of terminology has been established. It is not the purpose of this text to explain them in detail, but for the right introduction of the models, the sections will give definitions of the terms from the methodology's specification. [34]

1.4.3 Construction Model

The main component of the Construction model is Organization Construction Diagram (OCD) which shows the boundary of the process of the enterprise. It identifies transaction types, actor roles participating in the processes, and relations between them. [28]

DEMO Specification Language describes it as follows: *"The Construction Model (CM) of an organisation is the ontological model of its construction: the composition (the internal actor roles, i.e. the actor roles within the border of the organisation), the environment (i.e. the actor roles outside the border of the organisation that have interaction with internal actor roles), the interaction structure (i.e. the transaction kinds between the actor roles in the composition, and between these and the actor roles in the environment), and the interstriction structure (i.e. the information links from actor roles in the composition to internal transaction kinds and to external transaction kinds).*

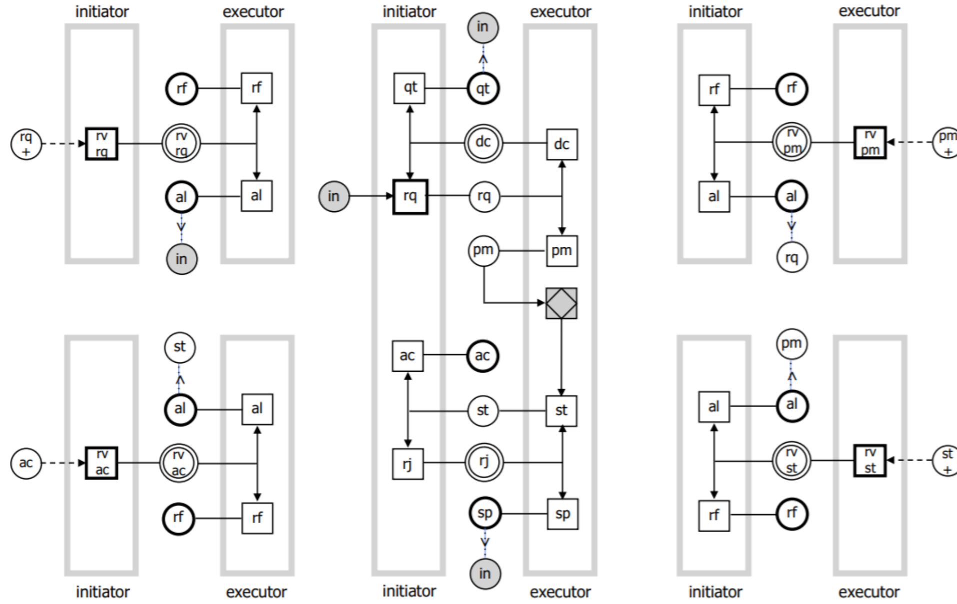
The CM of an organisation is represented in an Organisation Construction Diagram (OCD), a Transaction Product Table (TPT), and a Bank Contents Table (BCT)." [34]

1.4.4 Process Model

The Process Model describes the deep structure of the business processes in an enterprise. This model is also fits as a starting point for creating use cases. The part of it called the Process Structure Diagram (PSD) describes the process steps for every transaction and relationship between them of each process. [28] The Transaction Pattern Diagram (TPD) is a crucial component used in subsequent chapter and it is depicted in Figure 1.7.

DEMO Specification Language describes it as follows: *"The Process Model (PM) of an organisation is the ontological model of the state space and the transition space of its coordination world. Regarding the state space, the PM contains, for all internal and border transaction kinds, the process steps and the existence laws that apply, according to the complete transaction pattern. Regarding the transition space, the PM contains the coordination event kinds as well as the applicable occurrence laws, including the cardinalities of the occurrences. The occurrence laws within a transaction process are fully determined by the complete transaction pattern. Therefore, a PSD contains only the occurrence laws between transaction processes, expressed in links between*

Figure 1.7: Transaction Pattern Diagram[34]



process steps. There are two kinds: response links and wait links (which represent interventions).

A PM is represented in a Process Structure Diagram (PSD), optionally complemented by a Transaction Pattern Diagram (TPD) for one or more of the transaction kinds." [34]

1.4.5 Action Model

The Action Model specifies actions that correspond with situations outlined in the PSD. It contains specifications of action rules written in a pseudo-algorithmic language. [28]

DEMO Specification Language describes it as follows: *"The Action Model (AM) of an organisation organisation is the ontological model of its operation. It consists of a set of action rules and a set of work instructions. There is an action rule for every agendum kind for every internal actor role. An action rule specifies the (production and/or coordination) acts that must be performed, as well as the facts in the production world and/or the coordination world whose presence or absence in the state of the world must be assessed. Work instructions are optional. They guide the executor of a transaction in executing the production act.*

An AM is represented in Action Rule Specifications (ARS) and Work Instruction Specifications (WIS)." [34]

1.4.6 Fact Model

The Fact Model is partly the specification of the processed data and it is based on the Action Model. Its Object Fact Diagram declares result types and describes relations between object classes that occur in the processes. Formerly it was called the State Model. [28]

The current version of the DEMO Specification Language describes it as follows: *"The Fact Model (FM) of of an organisation is the ontological model of the state space and the transition space of its production world. Regarding the state space, the FM contains all identified fact kinds (both declared and derived), and the existence laws. Three kinds of existence laws are specified graphically: reference laws, unicity laws, and dependency laws; the other ones are specified textually. Regarding the transition space, the FM contains the production event kinds (results of transactions) as well as the applicable occurrence laws.*

The transition space of the production world is completely determined by the transition space of its coordination world. Yet it may be illustrative to show the implied occurrence laws in an OFD.

The FM is represented in an Object Fact Diagram (OFD), possibly complemented by Derived Fact Specifications and the (textual) Existence Law Specifications that cannot be expressed in the OFD." [34]

1.4.7 DEMO Machine

The DEMO methodology uses approaches that are harder to understand than other process modeling methods like Business Process Model and Notation (BPMN). The theoretical computation foundations were introduced to simplify access to this concept with the preservation of its expressiveness. [33]

The DEMO machine is such an abstract formalization of DEMO methodology suitable for software implementation. It describes the DEMO model as an ordered tuple consisting of an identifier, transaction kinds, actor kinds, conditional links, causal links, facts, and rules. The subsequent transaction then behaves according to this model. [33]

1.4.8 Current Solutions of Translation from BPM to SC

The graphic domain-specific language usage of DEMO Methodology was explored [35] but the translation from it for the Cardano blockchain was never implemented or analyzed.

Other solutions are using BPM as a description of the logic of the contract. Caterpillar compiler has chosen methodology BPMN, which is good for graphic process modeling. The models are then translated into the smart contract. This contract is written in the Solidity language of Ethereum blockchain. [36]

1.5 Chapter Summary

The chapter Theoretical Foundations laid base knowledge to realize the generation of the contract. It described all necessary components as a blockchain network where the smart contract will be executed, the Plutus Platform in which the smart contract will be written and the DEMO Methodology that will capture the model of the process that represents the logic behind the contract.

Bitcoin was a boom in cryptocurrencies, but the blockchain technology has bigger potential. Its advantages together with smart contracts in making processes more effective are undoubted, but the scale of deployment will be shown in practice.

Cardano blockchain has a unique approach to create its products by being particular about a research basis. This mitigates the potential risk in unexpected undesirable future use. Its programming language builds on Haskell which prevents many unintended side effects. The conclusion of the first goal, exploring the state-of-the-art Cardano, is that it is suited for running safe and effective smart contracts but the current absence of specific functionalities could be a limitation in generation from the DEMO models.

The models of the DEMO methodology well illustrates processes both in an enterprise and a contract. In the following chapter, the thesis will analyze the transformation into the smart contract.

Chapter 2

Plutus Smart Contract Generation from DEMO Models

In contrast with the previous chapter that laid out a theoretical basis, this chapter moves onto the practical part of exploring the generation of the smart contract itself. The first step will be an analysis of the technology's compatibility and conversion of DEMO models into a suitable object from which the smart contract is easily generated. In the second part, the chapter describes general ways the smart contract written in Plutus programming language could be generated.

Two approaches can be applied. The analysis can be done by taking and converting each of the DEMO models. The second approach starts with the general layout of the smart contract using the model currently needed. The chapter will begin by exploring how to represent each model in the smart contract as a whole and exploring the limitations of the representations. Then the analysis will focus on the generation mainly using the second approach with depictions of the output's pseudo-code.

2.1 DEMO and Extended UTXO Compatibility

The generation process, that this thesis is analyzing, has DEMO models on input represented as data structures and functioning Plutus Tx smart contract on the output. Because smart contracts written using the Plutus platform are domain-specific, they cannot describe reality as much as is possible in the DEMO methodology. The limitations of blockchain technology implicate that the generation will not utilize all parts of the DEMO models and all the practices of the methodology. Previous research about generation was related to Solidity and Etherum that differs in the accounting style base, and therefore the translation approach will differ too.

2.1.1 Domains of Smart Contracts

The technologies described in the previous chapter Theoretical Foundations enable the desired state replication. But with the combination of these technologies comes the limitations that make the blockchain network effective and reasonable only in specific situations. The blockchain despite basically being database is not suited as storage of unnecessary data. Its main domain is undeniably capturing history of ownership of the tokens. Tokens are meant any subject of value like coins of the cryptocurrencies or unique identification numbers representing real estates.

The blockchain's domain implies that the main action around which the processes are built consists of sending the token from one address to another. The logic applied to the sending utilizes virtual mediator. The validator script is that mediator and it splits the action into two: the payment of the token to the script's address and its collection.

2.1.2 Immutable Data in Validator Script

The extended UTXO allows introducing a condition or additional computation on the token's payment or collection. But the data supporting the condition are stored on-chain together with the script so it uses the same attributes as the tokens that they are immutable. The actualization of the data variables is also impossible by the nature of the Plutus programming language's functional paradigm and the idea of the blockchain. This creates an obstacle when implementing DEMO models in the Cardano.

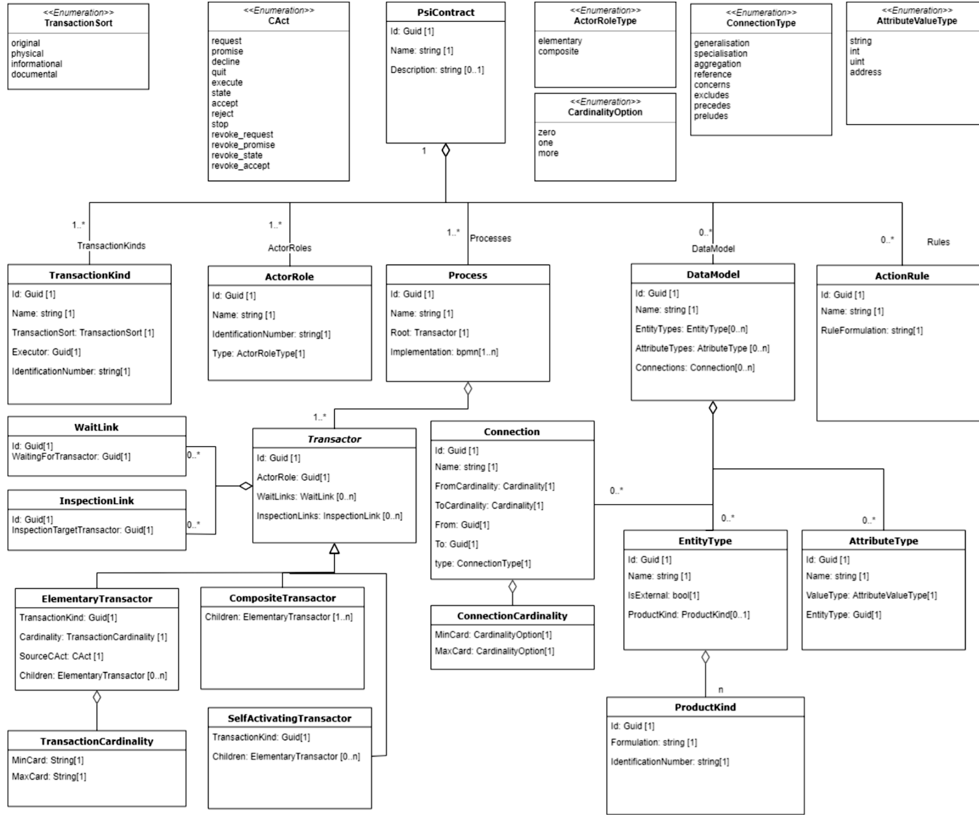
To represent the DEMO model in the smart contract, the different changing states in the validator script are needed. This requirement can be solved by using the libraries supporting the state machine. The concept of the state machine bypasses immutability by predefined states of the data script keeping the previous states still stored.

2.1.3 Contract Meta-Model

There are various ways to describe DEMO models. This thesis will base the input structure of the generation process on the meta-model that depicts the DEMO process models. The information and the logic declared in the model seen in Figure 2.1 will be the only source of the generation.

This thesis does not examine the right ways to model processes using the DEMO methodology. The meta-models used in generation by users could be the outcome of different modeling tools or derived from non-software formalization like DEMO Machine [33] which uses formalization and it is based on mathematical concepts.

Figure 2.1: Contract Meta-model: UML Diagram [37]



2.1.4 Representation of Construction Model

The Construction Model is the basic model that captures an outline of the process and addresses its actor types. The generation will use it as a stepping stone. The Organisation Construction Diagram is the more important part of the CM. The Transaction Product Table and the Bank Contents Table are not crucial for the generation. In the smart contract, they will be used for name definitions or for the logs informing the users.

The OCD of the Construction Model describes the relation between actor roles and transaction kinds. Actors' representation in the smart contract has two variations: actors controlled through the interface in the wallet and actors controlled by an automatic code. This distinction is clear from the elementary and composite type of actor role. The generation will utilize two different translation approaches for the variations.

The initiator and the executor are needed information that will be registered in the start step of specific DEMO transactions and then used for

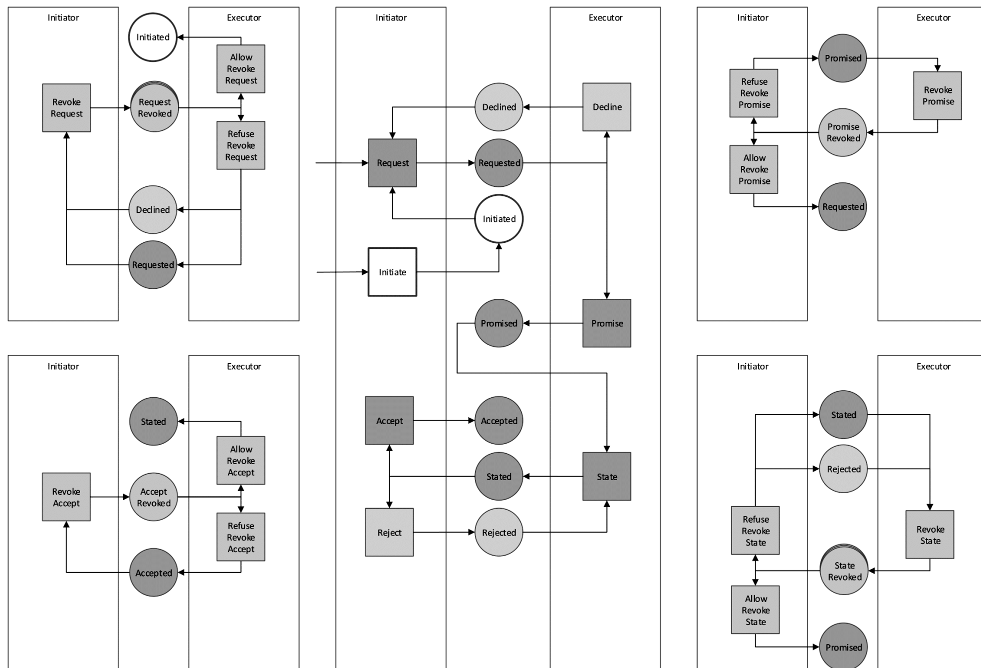
verification of the participators' access. The initiator of the whole process is derived from the self-generating attribute of the actor. Other self-generating actors will be able to request transaction kind without the context of other transactions that will be each represented by validator script and a state machine.

2.1.5 Representation of Process Model

The Process Model is essential for the generation and it describes elementary steps in the process. It also connects transaction kids between themselves as they progress. The structure of the Plutus' state machine will be based on the transaction axiom state machine from the DEMO Machine depicted in Figure 2.2. The Process Structure Diagram will serve for a description of the behavior depending on the context of other state machines.

The wallet's interface will be based on the coordination acts of the TPD. These endpoint functions will trigger the stepping of the state machine. The states are derived from coordination acts extended by the initial state.

Figure 2.2: Transaction Axiom State Machine[33]



The relations between the transactors from the Process Structure Diagram in the form of response and wait links portraying the dependence of one state machine on others. If the actor role is not represented by the public address

of the user, the response coordination acts will be sent by the internal code controlling conditional states each slot.

2.1.6 Representation of Fact Model

As a representation of the data structures used in the process, the Object Fact Diagram is used. It contains entities and their properties and attributes connected to others. The contract data model will represent these connections, types of entities or attributes, and product kinds. It also differentiates with attribute value types that are overlapping with Cardano blockchain value types. The other members of the FM like Derived Law Specifications and Existence Law Specifications are ignored.

The problem with implementing the FM is how to capture the flow of the data. The state made from a combination of constructors derived from coordination act and the data of that act is one possibility. Creating another type of the state machine in the contract where the changes of the process' data model are captured as a new state of its machine is another. There comes a problem with both of them. The state machine client returns state on request. The verification of a specific actor role that has access could be done off-chain but this violates contracts' security. The alternation of the client is needed to be implemented in the Plutus but in the following generation, the analysis will assume its existence.

2.1.7 Representation of Action Model

The AM specifies all action rules that serve as guides for actors on how to deal with coordination events. They are described the most loosely than any other model in DEMO. For a generation, there is a need for a pattern in actor rules. The rules of the smart contract will mostly define coordination actions but will be extended with payment and collection actions. The meta-model only defines the formulation of the action rules but the logic behind them is not described. The part of the generation where they are implemented is outlined in the endpoint function's template described later in this chapter.

2.2 Generation of On-Chain Code

The on-chain code of the smart contract generated from the DEMO models that run on the network itself consists mainly of the state machine definition, its instance, and the client. These concepts, like all constructs, are only functions in the functional paradigm so the thesis will use both terms. The two following sections will analyze all parts of the code with examples written in pseudo-code. There are also parts where the modules are imported and lines with pragmas instructing the GHC Haskell compiler.

2.2.1 Process Identification

There will be many running instances of the processes captured in the smart contract. When sending actions to the state machine, there is a need for a distinction of specific state machines. For that, the machine will be initialized together with an identification number unique for the process instance and shared between DEMO transactions. The part of the pseudo-code for the process identification can be seen in Figure 2.3.

Figure 2.3: Generated Pseudo-code: Process ID

```
1  data <ProcessName> = <ProcessName>
2      { id<ProcessName> :: ByteString
3      }
4      deriving (<Type1>, <Type2>, ... , <TypeN>)
5
6  makeLift ''<ProcessName>
```

2.2.1.1 Figure 2.3: Explanatory Notes

- The blockchain uses ByteString type to store strings for its space and time efficiency.
- The *deriving* keyword serves to inherit the behavior of other types.
- The data types stored on-chain needs to be lifted to the Plutus Core using *makeLift* keyword.

2.2.2 Data Type Representing Facts

The next part of the smart contract will be definitions of facts, their entities, and their attributes. This data type will be used as a part of the state for the state machine. It is necessary to consider implementing some of the facts outside the Cardano blockchain. The blockchain, in general, should store only necessary data because every node has a copy of all instances and could be storage heavy. The part of the pseudo-code for the representation of the facts can be seen in Figure 2.4.

2.2.2.1 Figure 2.4: Explanatory Notes

- The *data* keyword introduces new value with multiple possible constructors and multiple arguments.
- The data types that are used in states or data scripts, in general, need to be adjusted to Plutus Core using *makeIsData* keyword.

Figure 2.4: Generated Pseudo-code: Data Type Representing Facts

```
1  data Fact<FactName> = Fact<FactName>
2      { attr<AttributeName1>  ::  <AttributeType1>
3      , attr<AttributeName2>  ::  <AttributeType2>
4      ...
5      , attr<AttributeNameN>  ::  <AttributeTypeN>
6      }
7      deriving (<Type1>, <Type2>, ... , <TypeM>)
8
9  makeLift ''Fact<FactName>
10 makeIsData ''Fact<FactName>
```

2.2.3 Initiator and Executor Actor Roles

In the initialization of the state machines representing the Transaction Kinds, the initiator and executor roles have to be remembered. These roles will be referred by the public keys of the actors' wallets. If one of the roles is held by function's code then the role attribute is the address of the script. The data type containing the roles serves for verification when accessing the DEMO transaction from the wallet and it is used in a state of the transaction kinds' machine. The part of the pseudo-code for the data type that contains initiator and executor roles can be seen in Figure 2.5.

Figure 2.5: Generated Pseudo-code: Initiator and Executor Actor Roles

```
1  data Roles = Roles
2      { initiator :: PubKey
3      , executor :: PubKey
4      }
5      deriving (Prelude.Eq, Prelude.Show, Prelude.Ord,
           Generic, IotsType, ToSchema)
6
7  makeLift ''Roles
8  makeIsData ''Roles
```

2.2.4 States of Transaction Kinds' State Machines

The functions portraying the state machines of the Plutus programming language take two arguments: State and input. The states of the state machine representing transaction kinds refer to phases' locations in the TPD. The names are derived from the names of the coordination acts and they are extended by the initial state. The part of the pseudo-code for the states of the transaction kinds' state machines can be seen in Figure 2.6.

Figure 2.6: Generated Pseudo-code: States of the Transaction Kinds' State Machines

```
1  data TransactionKind<FactName>State =
2        Initial<FactName>   Roles
3      | Requested<FactName> Roles Fact<FactName>
4      | Promised<FactName>  Roles Fact<FactName>
5      | Stated<FactName>    Roles Fact<FactName>
6      | Accepted<FactName>  Roles Fact<FactName>
7      | Declined<FactName>  Roles Fact<FactName>
8      | Quitted<FactName>   Roles Fact<FactName>
9      | Rejected<FactName>  Roles Fact<FactName>
10     | Stopped<FactName>   Roles Fact<FactName>
11     deriving (Prelude.Eq, Prelude.Ord, Prelude.Show,
          Generic, IotsType)
12
13 makeLift ''TransactionKind<FactName>State
14 makeIsData ''TransactionKind<FactName>State
15
16 instance E.Eq TransactionKind<FactName>State where
17     (Initial<FactName> _)                       == (
          Initial<FactName> _)       =   True
18     (Requested<FactName> _ _)
          ==  (Requested<FactName> _ _)   =   True
19     (Promised<FactName> _ _)
          ==  (Promised<FactName> _ _)    =   True
20     (Stated<FactName> _ _)
          ==  (Stated<FactName> _ _)      =   True
21     (Accepted<FactName> _ _)
          ==  (Accepted<FactName> _ _)    =   True
22     (Declined<FactName> _ _)
          ==  (Declined<FactName> _ _)    =   True
23     (Quitted<FactName> _ _)
          ==  (Quitted<FactName> _ _)     =   True
24     (Rejected<FactName> _ _)
          ==  (Rejected<FactName> _ _)    =   True
25     (Stopped<FactName> _ _)
          ==  (Stopped<FactName> _ _)     =   True
26     _  ==  _                  =   False
```

2.2.4.1 Figure 2.6: Explanatory Notes

- The *instance* together with the *where* keyword implements equality function instance for the newly defined constructors.

2.2.5 Inputs of Transaction Kinds' State Machines

The second argument of the state machine is the input argument. The input argument is taken when the state machine is told to change a state. The names of the inputs correspond with the names of the coordination acts with the addition of initial input that gives the roles to the transaction kind. The complete transaction patter also contains the possibility to revoke previous actions. It is done similarly and it is omitted. The part of the pseudo-code for the inputs of the transaction kinds' state machines can be seen in Figure 2.7.

Figure 2.7: Generated Pseudo-code: Inputs of the Transaction Kinds' State Machines

```
1  data TransactionKind<FactName>Action =
2        Initiate<FactName> Roles
3      | Request<FactName> Fact<FactName>
4      | Promise<FactName>
5      | State<FactName>
6      | Accept<FactName>
7      | Decline<FactName>
8      | Quit<FactName>
9      | Reject<FactName>
10     | Stop<FactName>
11     deriving (Prelude.Eq, Prelude.Ord, Prelude.Show,
          Generic, IotsType)
12
13 makeLift ''TransactionKind<FactName>Action
14 makeIsData ''TransactionKind<FactName>Action
```

2.2.6 Transition Functions of Transaction Kinds' State Machines

The state machine data type has three members. The first of them is a transition function that takes a current state and an input and returns a new state. The names of the output state will mostly correspond with the names of the input's coordination acts but in the states: Requested, Declined, Stated and Rejected, there are two choices. The roles of the initiator and executor are taken from the previous state. The part of the pseudo-code for the transition functions of the transaction kinds' state machines can be seen in Figure 2.8.

2.2.6.1 Figure 2.8: Explanatory Notes

- The *Maybe* data type returns value *Just a* where *a* is some other type or *Nothing*.

Figure 2.8: Generated Pseudo-code: Transition Functions of the Transaction Kinds' State Machines

```
1  transition<TxKN> :: <ProcessName> ->    TransactionKind
       <FactName>State -> TransactionKind<FactName>Action
       -> Maybe TransactionKind<FactName>State
2  transition<TxKN> _ (Initial<FactName> ie)      (Request
       <FactName> fct) = Just (Requested<FactName> ie fct)
3  transition<TxKN> _ (Requested<FactName> ie fct) Decline
       <FactName> = Just (Declined<FactName> ie fct)
4  transition<TxKN> _ (Requested<FactName> ie fct) Promise
       <FactName> = Just (Promised<FactName> ie fct)
5  transition<TxKN> _ (Declined<FactName> ie fo)  (Request
       <FactName> fct) = Just (Requested<FactName> ie fct)
6  transition<TxKN> _ (Declined<FactName> ie fct)     Quit
       <FactName> = Just (Quitted<FactName> ie fct)
7  transition<TxKN> _ (Promised<FactName> ie fct)    State
       <FactName> = Just (Stated<FactName> ie fct)
8  transition<TxKN> _ (Stated<FactName> ie fct)     Reject
       <FactName> = Just (Rejected<FactName> ie fct)
9  transition<TxKN> _ (Stated<FactName> ie fct)     Accept
       <FactName> = Just (Accepted<FactName> ie fct)
10 transition<TxKN> _ (Rejected<FactName> ie fct)    State
       <FactName> = Just (Stated<FactName> ie fct)
11 transition<TxKN> _ (Rejected<FactName> ie fct)     Stop
       <FactName> = Just (Stopped<FactName> ie fct)
12 transition<TxKN> _ _ _ = Nothing
```

- The *TxKN* is a shortcut for Transaction Kind Name and was selected to streamline comprehensive examples.

2.2.7 Check Functions of State Machines

The second member of the state machine is the check function. The check function adds conditions considering the data type. In the generation from the DEMO is not necessary because its data type in validator script contains only the id of the process instance. The part of the pseudo-code for the check functions of the state machines can be seen in Figure 2.9. Its structure is the same for both state machines.

2.2.8 Final Functions of State Machines

The last member of the state machine definition is the final function. It simply returns if the current state is final or not. It is also universal for all transaction kinds. The part of the pseudo-code for the check functions of the state machines can be seen in Figure 2.10

Figure 2.9: Generated Pseudo-code: Check Functions of the State Machines

```
1  check<TxKN> :: <ProcessName> -> TransactionKind<
      FactName>State -> TransactionKind<FactName>Action ->
       PendingTx -> Bool
2  check<TxKN> _ _ _ _ = True
3  check<TxKN> _ (Initial<FactName> roles)        (Request
      <FactName> _) penTx = txSignedBy penTx (ini roles)
4  check<TxKN> _ (Declined<FactName> roles _)     (Request
      <FactName> _) penTx = txSignedBy penTx (ini roles)
5  check<TxKN> _ (Declined<FactName> roles _)         Quit
      <FactName> penTx = txSignedBy penTx (ini roles)
6  check<TxKN> _ (Stated<FactName> roles _)        Accept
      <FactName> penTx = txSignedBy penTx (ini roles)
7  check<TxKN> _ (Stated<FactName> roles _)        Reject
      <FactName> penTx = txSignedBy penTx (ini roles)
8  check<TxKN> _ (Requested<FactName> roles _)     Promise
      <FactName> penTx = txSignedBy penTx (exe roles)
9  check<TxKN> _ (Requested<FactName> roles _)     Decline
      <FactName> penTx = txSignedBy penTx (exe roles)
10 check<TxKN> _ (Promised<FactName> roles _)        State
      <FactName> penTx = txSignedBy penTx (exe roles)
11 check<TxKN> _ (Rejected<FactName> roles _)        State
      <FactName> penTx = txSignedBy penTx (exe roles)
12 check<TxKN> _ (Rejected<FactName> roles _)         Stop
      <FactName> penTx = txSignedBy penTx (exe roles)
```

Figure 2.10: Generated Pseudo-code: Final Functions of the State Machines

```
1 isFinal<TxKN> :: TransactionKind<FactName>State -> Bool
2 isFinal<TxKN> (Quitted<FactName> _ _)    =   True
3 isFinal<TxKN> (Stopped<FactName> _ _)    =   True
4 isFinal<TxKN> (Accepted<FactName> _ _)   =   True
5 isFinal<TxKN> _
                     =   False
```

2.2.9 Definition of State Machines

In this part of the on-chain code generation, it is needed to define a state machine. But this definition does not suffice on itself. The code needs to be assigned to a specific validator script and compiled to Plutus Core. The part of the pseudo-code for the definition of the transaction kinds' state machines can be seen in Figure 2.11.

Figure 2.11: Generated Pseudo-code: Transaction Kinds' State Machines

```
1  machine<TxKN> :: <ProcessName> -> StateMachine
       TransactionKind<FactName>State TransactionKind<
       FactName>Action
2  machine<TxKN> pn = StateMachine
3      { smTransition  =   transition<TxKN> pn
4      , smCheck       =   check<TxKN> pn
5      , smFinal       =   isFinal<TxKN>
6      }
7
8  validator<TxKN> :: <ProcessName> -> ValidatorType (
       StateMachine TransactionKind<FactName>State
       TransactionKind<FactName>Action)
9  validator<TxKN> pn = mkValidator (machine<TxKN> pn)
10
11 script<TxKN> :: <ProcessName> -> ScriptInstance (
       StateMachine TransactionKind<FactName>State
       TransactionKind<FactName>Action)
12 script<TxKN> pn =
13     let val = $$(compile [|| validator<TxKN> ||])
14             `applyCode`
15                 liftCode pn
16         wrap = wrapValidator @TransactionKind<FactName>
               State @TransactionKind<FactName>Action
17     in validator @(StateMachine TransactionKind<
           FactName>State TransactionKind<FactName>Action)
18         val $$(compile [|| wrap ||])
```

2.2.10 Transaction Kinds' State Machines Instance and Client

The last part of the on-chain code is the definition of the state machine instances and clients. The clients support many useful functions to operate with state machines more easily. The part of the pseudo-code for the definition of the transaction kinds' state machine instances and clients can be seen in Figure 2.12.

2.3 Generation of Off-Chain Code

In the second part of the smart contract's generation, the focus is set on the off-chain code. This code runs only on the wallet and it is the weakest point of security. In the current state of the Plutus programming language, many components to generate smart contracts from the DEMO methodology are missing and have to be revised in the future. The generation assumes that

Figure 2.12: Generated Pseudo-code: Transaction Kinds' State Machines Instance and Client

```
1  instance<TxKN> :: <ProcessName> -> StateMachineInstance
       TransactionKind<FactName>State TransactionKind<
      FactName>Action
2  instance<TxKN> pn = StateMachineInstance
3      { stateMachine = machine<TxKN> pn
4      , validatorInstance = script<TxKN> pn
5      }
6
7  allocate<TxKN> :: TransactionKind<FactName>State ->
      TransactionKind<FactName>Action -> Value ->
      ValueAllocation
8  allocate<TxKN> _ _ currentValue =
9      ValueAllocation
10         { vaOwnAddress = currentValue
11         , vaOtherPayments = mempty
12         }
13
14 client<TxKN> :: <ProcessName> -> StateMachineClient
      TransactionKind<FactName>State TransactionKind<
      FactName>Action
15 client<TxKN> pn = mkStateMachineClient (instance<TxKN>
      pn) allocate<TxKN>
```

some features will be implemented in the future, for example, logging that was deleted because of revisions of the Wallet module.

2.3.1 Parameters of Endpoint Functions

The parameters are data types from which the on-chain data are then filled. They cannot be lifted to the Plutus Core that is why the existing custom data types cannot be used. The part of the pseudo-code for the definition of the parameters of the endpoint functions can be seen in Figure 2.13.

2.3.2 Schema Definition

The schemes are used for the wallet to render buttons of the interface and demanding the user for necessary parameters. It is necessary to define it before the functions are used in it. The part of the pseudo-code for the definition of the schema can be seen in Figure 2.14.

Figure 2.13: Generated Pseudo-code: Parameters of the Endpoint Functions

```
1  data Parameters<TxKN><CAct> = Parameters<TxKN><CAct>
2      { param<AttributeName1>  ::  <AttributeType1>
3      , param<AttributeName2>  ::  <AttributeType2>
4      ...
5      , param<AttributeNameN>  ::  <AttributeTypeN>
6      }
7      deriving (<Type1>, <Type2>, ... , <TypeM>)
8
9  makeLift ''Parameters<TxKN><CAct>
10 makeIsData ''Parameters<TxKN><CAct>
```

Figure 2.14: Generated Pseudo-code: Schema Definition

```
1  type Schema = BlockchainActions
2      .\/ Endpoint "<Transaction_Kind_Name_Action_1>"
          Parameters<TxKN1><CAct>
3      .\/ Endpoint "<Transaction_Kind_Name_Action_2>"
          Parameters<TxKN2><CAct>
4      ...
5      .\/ Endpoint "<Transaction_Kind_Name_Action_M>"
          Parameters<TxKNM><CAct>
6
7  mkSchemaDefinitions ''Schema
```

2.3.3 Endpoint Coordination Functions

The endpoint functions that deal with coordination acts from the users. A lot of the DEMO logic is implemented here but should be moved on-chain for security reasons when the technology allows that in the future. Aside from response and wait links from the PSD positioned here using conditions. The roles of actors of the initiator and executor could be represented by the code. The logic of the code is realized in this function. The methods of the state machine client used here are: *runInitialise* to initial setting of the state machine, *runStep* to move machine to another state, and *getOnChainState* to get facts or states used in conditions.

This is another weak area for security because computing is partly done in the node's wallet. Because this solution won't be used in the practical generation of this will be only an outline but not examined in detail. The same relates to the application of the action rules that would be implemented in this part and the logic of the automatic code represented by the event handler. The outlined part of the pseudo-code for the definition of the endpoint functions can be seen in Figure 2.15.

Figure 2.15: Generated Pseudo-code: Endpoint Functions

```
1  do<TxKN><CAct> :: (AsContractError e
2      , AsSMContractError e TransactionKind<FactName1>
          State TransactionKind<FactName1>Action
3      , AsSMContractError e TransactionKind<FactName2>
          State TransactionKind<FactName2>Action
4      ...
5      , AsSMContractError e TransactionKind<FactNameN>
          State TransactionKind<FactNameN>Action
6      ) => Contract Schema e ()
7  do<TxKN><CAct> = do
8      params <- endpoint @"do<TxKN><CAct>" @Parameters<
          TxKN><CAct>
9      let pn = <ProcessName> (param<TxKN><CAct>Id<
          ProcessName> params)
10     <TxKInitializationFunction1>
11     <TxKInitializationFunction2>
12     ...
13     <TxKInitializationFunctionM>
14 if <Condition1> && <Condition2> && ... && <ConditionK>
15         then
16             <TxKResponseFunction1>
17             <TxKResponseFunction2>
18             ...
19             <TxKResponseFunctionL>
20             --
21             <TokenActionFunction1>
22             <TokenActionFunction2>
23             ...
24             <TokenActionFunctionJ>
25         else
26             pure ()
```

2.3.4 Error Instances of State Machines

It is necessary to implement new errors on which wallet responds for each of the created state machines. This implementation causes problems with multi-parameter type classes that are necessary for the usage of state machines. The part of the pseudo-code for the definition of the error instances of the state machine can be seen in Figure 2.16.

2.3.5 Registration of Endpoints

The final part of the whole code is a registration of the endpoint function to the Plutus Playground interface. We need to distinct endpoint functions available to the users and the supporting functions. The part of the pseudo-

code for the definition of the registration of the wallets' endpoints can be seen in Figure 2.17.

Figure 2.16: Generated Pseudo-code: Endpoint Functions

```
1  data ErrorTransactionKind<FactName> =
2      ContractErrorTransactionKind<FactName>
           ContractError
3      | SMErrorTransactionKind<FactName> (SMContractError
            TransactionKind<FactName>State TransactionKind<
           FactName>Action)
4      deriving (Show)
5
6  makeClassyPrisms ''ErrorTransactionKind<FactName>
7
8  instance AsContractError ErrorTransactionKind<FactName>
       where
9      _ContractError
10         = _ContractErrorTransactionKind<FactName>
11
12 instance AsSMContractError ErrorTransactionKind<
      FactName> TransactionKind<FactName>State
      TransactionKind<FactName>Action where
13     _SMContractError
14         = _SMErrorTransactionKind<FactName>
```

Figure 2.17: Generated Pseudo-code: Registration of the Endpoints

```
1  endpoints :: (AsContractError e
2      , AsSMContractError e TransactionKind<FactName1>
           State TransactionKind<FactName1>Action
3      , AsSMContractError e TransactionKind<FactName2>
           State TransactionKind<FactName2>Action
4      ...
5      , AsSMContractError e TransactionKind<FactNameN>
           State TransactionKind<FactNameN>Action
6      ) => Contract Schema e ()
7  endpoints = <FunctionName1> <|> <FunctionName2> <|> ...
       <|> <FunctionNameM>
```

2.4 Chapter Summary

In this chapter, it was examined the analysis of the generation of the Plutus smart contracts from the DEMO process models. It provides various examples of pseudo-code capturing parts of the smart contract that will be used as templates in the implementation.

The Cardano blockchain differs from other networks by having its programming language for the smart contracts built in the principles of functional paradigms. This narrows the options to represent DEMO models because the data stored on-chain are immutable. The solution to this limitation is the usage of the state machines.

The examples of the smart contract's part are divided into on-chain and off-chain parts. It was discovered that a considerable part of the code capturing the logic of the DEMO methodology has to be moved to the off-chain section. This brings big security problem and the state machines need to be adjusted before the use of the generation in practice. Despite the presented obstacles, the generation of the Plutus smart contracts is successful and the outlined pseudo-code examples will solve as templates in the implementation.

Chapter 3

Proof of Concept

The last chapter of the thesis implements the actual generator of Plutus smart contracts from DEMO models. The result is then tested and simulated on a use-case describing one part of the simplified land title recording process.

3.1 Used Technologies

In addition to the tools of the Plutus Platform, the implementation utilizes technologies for application development and templating languages. The program for generating smart contracts is built as an extension of a basic templating processor (or engine). In general, templating processors combine templates and a data model into a functional desired code. Various templating engines are providing their own templating languages. The Fluid templating engine was used in this proposed solution.

3.1.1 .NET Core

In general, .NET technology is a cross-platform for developing applications for the web, mobile technology, desktop computers and the Internet of Things. It is an open-source project from the independent organization .NET Foundation. It provides multiple languages, tools, and libraries supporting software development.

.NET Core is an implementation of .NET for websites, servers, and a console application that runs on Windows, Linux and macOS operating systems. [38] It uses the same standard as other .NET implementations which allows utilization of the common APIs.

3.1.2 Liquid Language

Liquid is an open-source project developed by e-commerce company Shopify Incorporated. [39] It is both templating languages that have their own syntax

without a concept of a state. The designer doesn't need to know data content so it can be applied to multiple stores. It combines own Liquid language with HTML. Liquid also allows manipulating data in a template by using filters and the use of logic in a template.

3.1.3 Fluid Templating Engine

Fluid is a templating processor rendering and parsing Liquid templates (with additional extensions). It is developed as open-source software by Microsoft's software development engineer Sébastien Ros. [40] Another useful feature is registering .NET types and proprieties that are used in the data model displayed in a template. The Fluid is also better in performance than other processors generating Liquid templates such as DotLiquid or Liquid.NET.

3.1.4 Plutus Playground

Plutus Playground is an application of the Plutus platform for the GHC compilation of the code written in Plutus Tx. The output can be then simulated without real blockchain. The online version can be found on the Cardano testnet. The implementation of this thesis used the local version from Github repository [22]. This version was updated in September 2019 and it brings necessary components for this thesis but lacks others like logging. Because of that, the output of the generator will be compiled but not simulated on Plutus Playground.

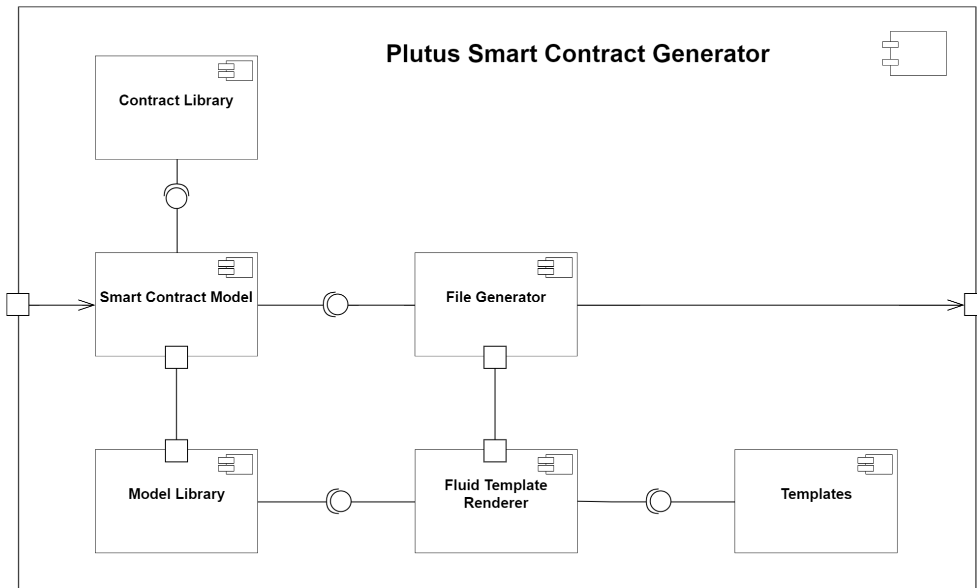
3.2 Software Architecture

The implementation that realizes actual generation from DEMO process models to the smart contract written on the Plutus platform is divided into multiple components. Aside from supporting classes and templates introduced in the previous chapter, the crucial parts are the File Generator, the Fluid Template Renderer, and the Smart Contract Model. The organization of the components can be seen in Figure 3.1. The implementation also contains test classes with the use-case which is described later.

3.2.1 Contract Classes

The Contract Library contains classes describing DEMO models captured by the contract meta-model. This model was slightly altered (for example more value types) to purposes of Cardano and this thesis. The specific model should be delivered using a modeling tool. The test class contains a model derived from one use-case implementation and it was filled manually.

Figure 3.1: Component Diagram of the Implementation



3.2.2 Smart Contract Model

The Smart Contract Model utilizes classes defined in the Model Library that is used as the data model for the template processor. It also provides methods that transform the contract model created from DEMO process models to the smart contract model. The structures that represent functions also use the Fluid Template Renderer described below to prerender functions then filling the main template.

3.2.3 File Generator and Fluid Template Renderer

The output of the File Generator is the actual compile-able smart contract with suffix .hs for Haskell codes. It saves the file with the given path and a file name. First, it registers custom classes and structures unknown to the Fluid templating engine. Then it passes the smart contract model and a string of final template to the Fluid Template Renderer that combines both objects.

3.3 Testing

The generator also implements a test class that contains the contract model derived from the DEMO models describing one use case from the actual process. The process used and described below is called land title recording. Specifically, the thesis is describing the transfer part of the process because it

is different from the ones like initial appropriation or splitting the land title into multiple ones.

3.3.1 Land Title Transfer Recording

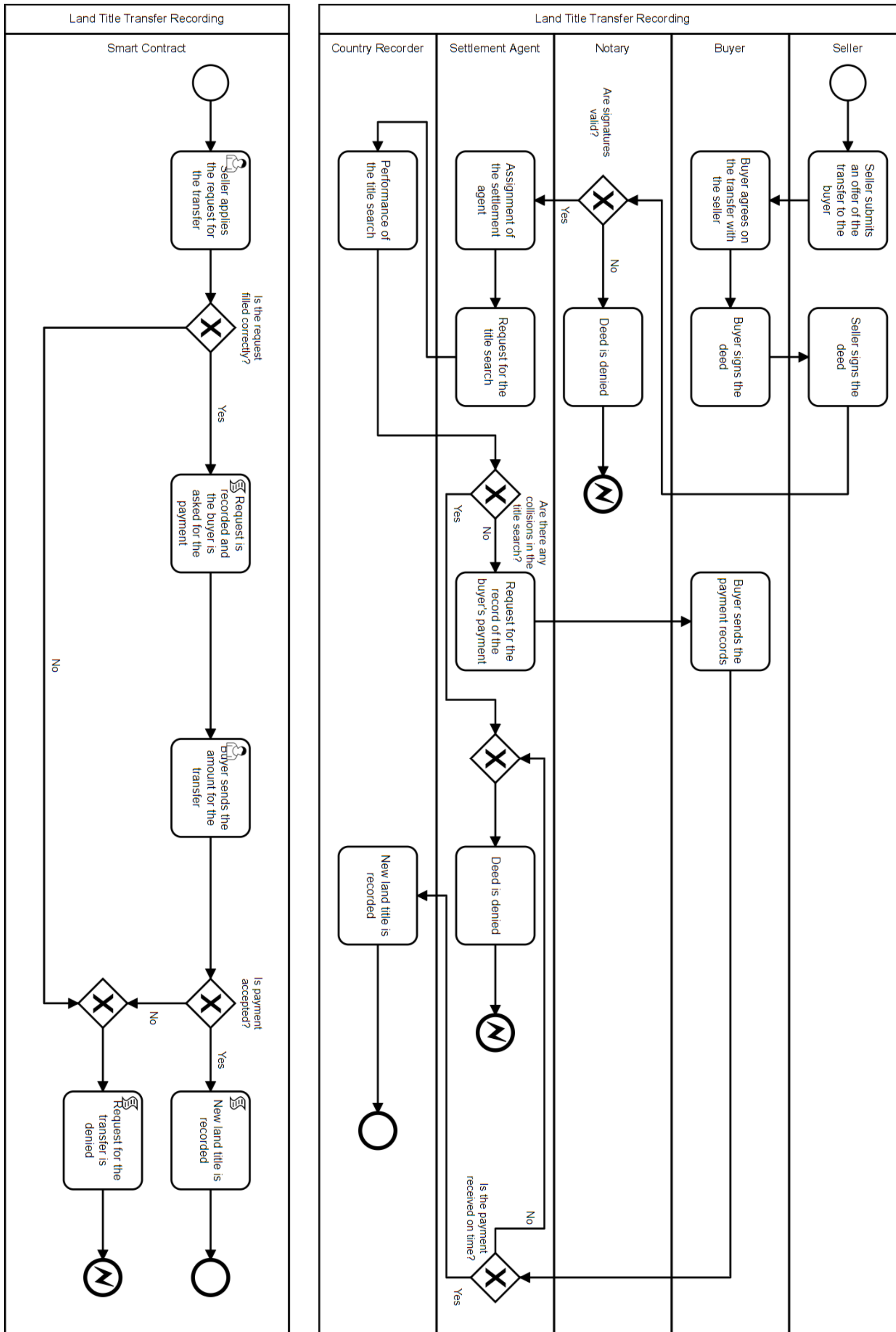
The following outline of the process of land title recording is just one of many that governments introduced for knowing on which subjects' property to collect taxes. In 2017, Cook Country recorder of deeds issued a report that explored the adoption of blockchain in real estate and the conclusions showed a success. [41] In the present times, when all land in the country is divided between state and private subjects, the transfer is the most common recording of the land title.

3.3.2 Steps of Land Title Transfer Recording

The following steps of a land title transfer recording are simplified to the basic thoughts behind the process. There are too many law systems of different countries that deal with several theoretical bases. Their analysis is not the subject of this thesis. The process described below was based on the multiple definitions in the Legal Dictionary of the Free Dictionary. [42]

> **Step 1** – After a buyer and a seller agree on the subject and price of the transaction, the notary checks the validity of the transaction and confirms it by his authority. In this case, the transaction is called the deed. A deed is a verified document conveying an interest in real property.
>
> **Step 2** – In this step, the settlement agent get in. The settlement agent is responsible to oversee that all laws and regulations are followed in transferring real estate. Mainly he performs title search: Research done to trace a title back to its original owner to ensure that there are no collisions with other claims. He does this by contacting the county recorder for records of the real estate.
>
> **Step 3** – When the title search is successfully done, the buyer is informed to submit verification of the payment for the property. After this, the settlement agent notifies the county recorder to change his records and the transaction between buyer and seller is completed.

Figure 3.2: Land Title Transfer Recording: Processes' Approaches Compared



3.3.3 Process Captured by Smart Contract

The previously described process is captured in Figure 3.2 together with its version using a smart contract which is much simpler. The complexity made up mainly of administration is necessary from the need to archive evidence that can later serve as a proof in a contractual dispute about the land title. The thesis showed an example that doesn't require any other authority. The state will probably demand a presence and control of another actor, which could make the process different.

3.3.4 DEMO Models of Land Title Transfer Recording

This section shows parts of DEMO models that the generation uses in the forming of the meta-contract. The OCD is described in Figure 3.3, the PSD in Figure 3.4 and the OFD in Figure 3.5. the thesis also outlines one of the Action Rule in Figure 3.6.

Figure 3.3: Land Title Transfer Recording: OCD

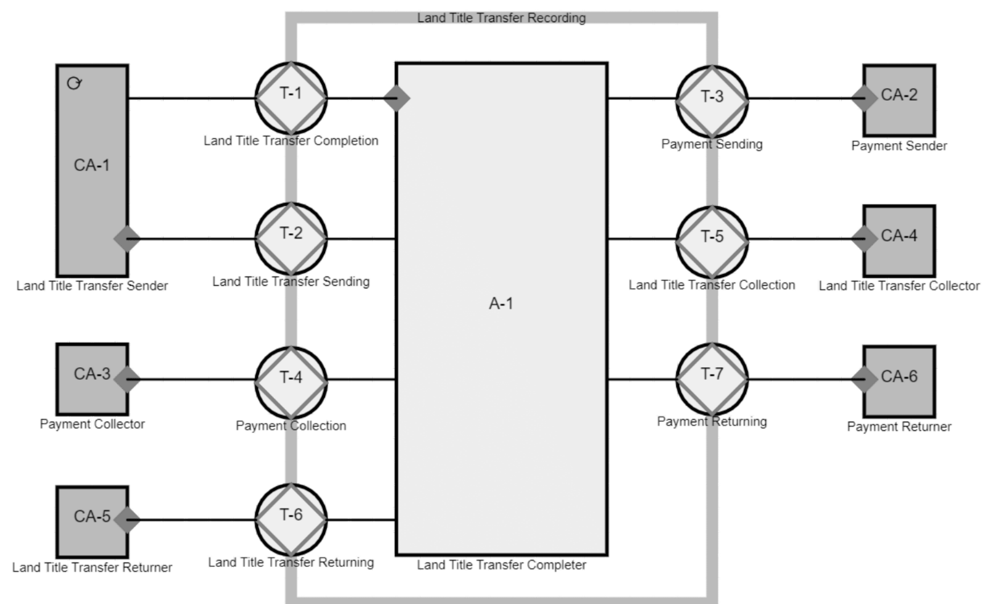


Figure 3.4: Land Title Transfer Recording: PSD

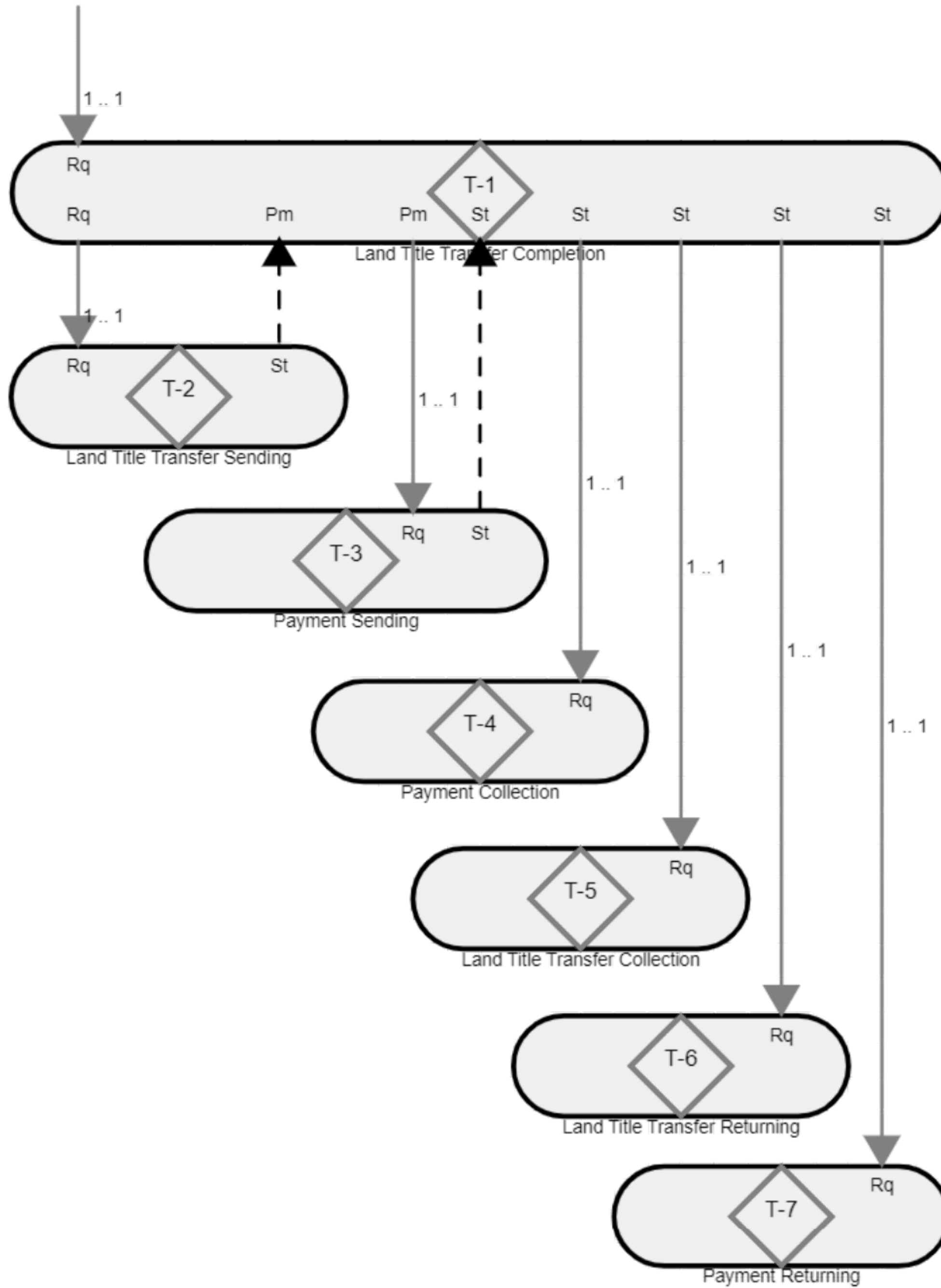


Figure 3.5: Land Title Transfer Recording: OFD

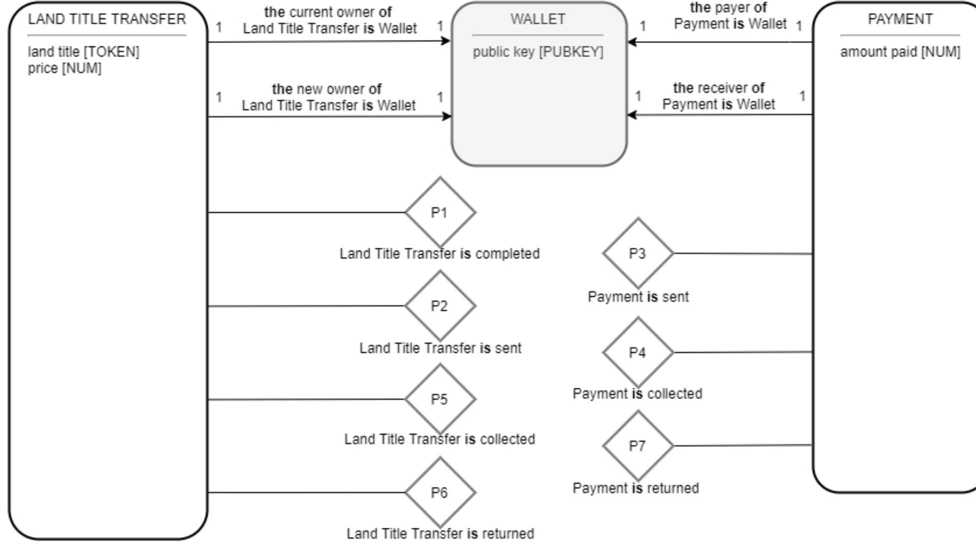


Figure 3.6: Land Title Transfer Recording: Action Rule Resolving Transfer

when	land title transfer completion **for** Land Title Transfer <u>is stated</u>		(T1/st)
assess	*justice:*	**the** performer **of the** state **is the** land title transfer completer **of** Land Title Transfer	
	sincerity:	< no specific condition >	
	truth:	**the** amount paid **of** Payment **is equal or greater than the** price **of** Land Title Transfer	
if	*complying with state is considered justifiable*		
then	<u>request</u>	land title transfer collection **for** Land Title Transfer	[T4/rq]
	<u>request</u>	payment collection **for** Payment	[T5/rq]
else	<u>request</u>	land title transfer returning **for** Land Title Transfer	[T6/rq]
	<u>request</u>	payment returning **for** Payment	[T7/rq]

3.3.5 Generated Code

The implementation generates a correct smart contract that is comprehensive and the Plutus Playground is not able to compile it as a whole. When dividing it into multiple parts the code is successfully compiled.

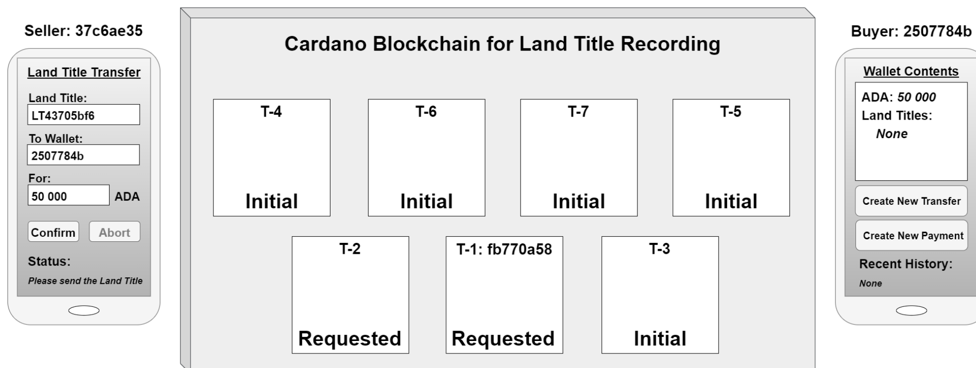
3.3.6 Simulation

Because of the division mentioned in the previous section, the simulation in the Plutus Playground is not possible. The lack of wallet logs and incompatibility of the state machine error in the type signature of the endpoint functions makes Plutus Playground unfit for simulation. In spite of the simulation not being supported by the Plutus Playground and not being the specified goal of this thesis, it will be done at least manually.

3.3.6.1 Initializing Land Title Transfer Recording

Figure 3.7 shows the starting point of the land title transfer recording. The seller owns a token representing the land title and wants to exchange it with another person. He initializes all state machines representing transactions and requests the Land Title Transfer Completion (T-1).

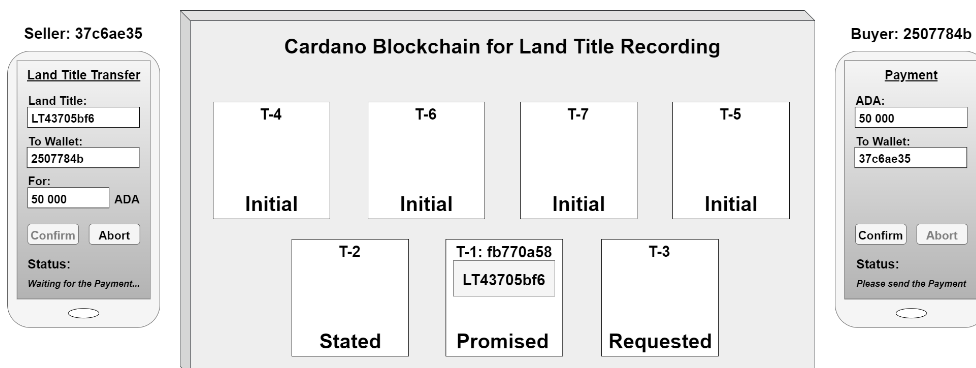
Figure 3.7: Simulation: Initializing the Land Title Transfer Recording



3.3.6.2 Sending Land Title Transfer

The Land Title Transfer Completion (T-1) moves to the promised state only after the seller sends Land Title Transfer containing all necessary information and the token. Figure 3.8 depicts the part where the seller paid the token together with the Land Title Transfer to the T-1's address and the state machine portraying the Land Title Transfer Sending (T-2) is stated.

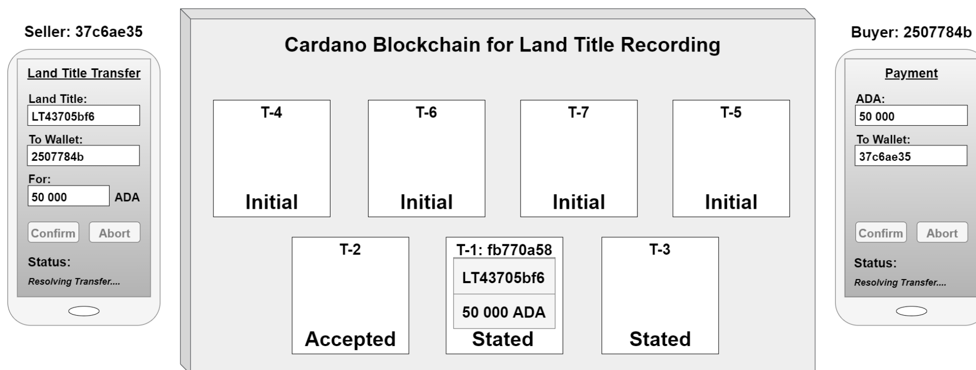
Figure 3.8: Simulation: Sending the Land Title Transfer



3.3.6.3 Sending Payment

In the part where the Land Title Transfer was sent and the T-1 moved to the promised state, the Payment Sending (T-3) was requested. After the buyer paid the specific amount ADAs, the T-3 moves to the stated state as seen in Figure 3.9.

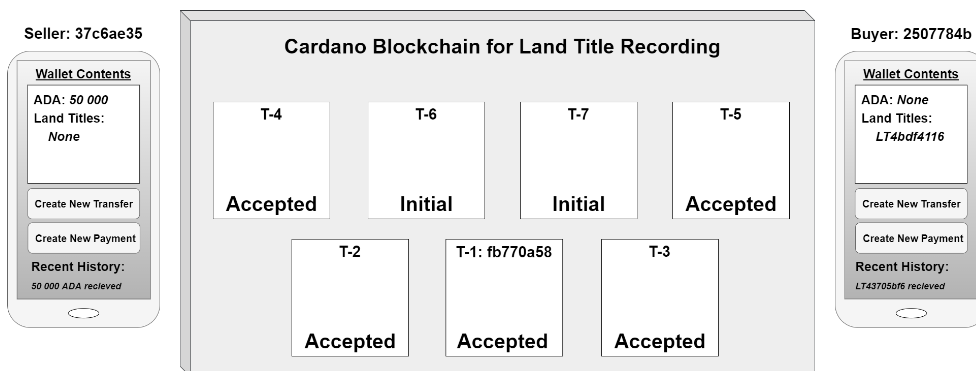
Figure 3.9: Simulation: Sending the Payment



3.3.6.4 Resolving Land Title Transfer Recording

The last Figure 3.10 shows the resolution of the transfer. The amount specified in the Land Title Transfer is compared and if it is equal or greater than the amount in the Payment, the land title's token is sent to the buyer and the value of the Payment is sent to the seller. If the condition is not fulfilled, the T-6 the T-7 are requested instead of the T-4 and the T-5 and the result is that the tokens are returned to the former owners.

Figure 3.10: Simulation: Resolving the Land Title Transfer Recording



3.4 Chapter Summary

The final chapter exploring the generation of the Plutus smart contracts to the DEMO process models introduced an implementation. Then it tested the use-case of the land title recording. In the first part of the chapter, the implementation was described from different point of views. The used technologies were summarized and the section explored the implementation's organization of the components.

Aside from testing, the second part of this chapter provided examples of DEMO models depicting the land title transfer recording. The visual simulation outlined the usefulness of such a process and the possibilities of such a concept. The specific solution of the recording can be extended using revokes or time limits. The same principle will also apply to any exchange of other types of tokens.

Conclusion

The goal of this thesis was to describe way to generate process models from DEMO to a smart contract written in Plutus (specifically Plutus Tx). Previously to this description, Cardano and Plutus technologies were explored to decide whether they were fit for generating such contracts. Finally, after providing this information, proof of concept implementation was presented.

Cardano is a promising platform for technology with a working blockchain supporting ADA cryptocurrency and its Plutus programming language. It bases all development on precise research backed up by many professional publications. Some aspects still need to be considered as the whole project is unfinished. Plutus Tx is a very useful programming language for smart contracts because its functional paradigm reduces fault-creation and increases efficiency.

To capture DEMO process models with the Plutus Platform, the use of state machines was chosen as the proper way to generate smart contracts. Despite limitations of the state machines' interface that is solvable, the devised procedure using templates for the process of generation was successful. Overall, the final generation of smart contracts will significantly simplify their construction thus reducing costs and time on the development of a blockchain.

The presented approach was then implemented using .NET Core. As a method to transform models to code, a template engine was used. The advantages of this generation model are clearly visible in the comparison of the current land title recording and one achieved using the smart contract.

For future research, this thesis submits and makes a proposal of extending the capabilities of state machines by implementing more logic on-chain. The generation described in this thesis could be also just a part of a larger tool for the generation of smart contracts. Tools designed for specific DEMO process models can be a useful and efficient way to support such a generation. Plutus is also only one of many platforms that could be explored.

Bibliography

[1] World Bank Group. Time Required to Enforce a Contract (Days). [online], [2019-12-09]. Available from: https://data.worldbank.org/indicator/IC.LGL.DURS

[2] Hornáčková, B. *Using Blockchain Smart Contracts in the DEMO Methodology*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2018.

[3] Wattenhofer, R. *Blockchain Science: Distributed Ledger Technology*. Inverted Forest Publishing, third edition, 2019, ISBN 978-1793471734, 289 pp.

[4] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. [online], 2008, [2019-12-09]. Available from: https://bitcoin.org/bitcoin.pdf

[5] Lipovyanov, P. *Blockchain for Business 2019: A user-friendly introduction to blockchain technology and its business applications*. Birmingham: Packt Publishing, 2019, ISBN 978-1789956023.

[6] Jimi, S. How does blockchain work in 7 steps — A clear and simple explanation. [online], 2018, [2019-12-09]. Available from: https://blog.goodaudience.com/blockchain-for-beginners-what-is-blockchain-519db8c6677a

[7] Maldonado, F. C. *Introduction to Blockchain and Ethereum: Use distributed ledgers to validate digital transactions in a decentralized and trustless manner*. Birmingham: Packt Publishing, 2018, ISBN 978-1789612714.

[8] Brünjes, L.; Vinogradova, P. *Plutus: Writing reliable smart contracts*. Input Output HK, 2019. Available from: https://leanpub.com/plutus-smart-contracts

[9] Lamport, L.; Shostak, R.; Pease, M. The Byzantine Generals Problem. [online], 1982, [2019-12-09]. Available from: https://people.eecs.berkeley.edu/~luca/cs174/byzantine.pdf

[10] Jakobsson, M.; Juels, A. Proofs of Work and Bread Pudding Protocols. [online], 1999, [2019-12-09]. Available from: http://www.hashcash.org/papers/bread-pudding.pdf

[11] Bitcoin Core. Bitcoin Core: Bitcoin. [online], [2019-12-09]. Available from: https://bitcoincore.org

[12] Bashir, I. *Mastering Blockchain: Deeper insights into decentralization, cryptography, Bitcoin, and popular Blockchain frameworks*. Birmingham: Packt Publishing, 2017, ISBN 978-1787125440.

[13] Szabo, N. Smart Contracts Glossary. [online], 1995, [2019-12-09]. Available from: https://nakamotoinstitute.org/smart-contracts-glossary/

[14] Szabo, N. Multinational Small Business. [online], 1993, [2019-12-09]. Available from: https://nakamotoinstitute.org/multinational-small-business/

[15] Clack, C. D.; Bakshi, V. A.; Braine, L. Smart Contract Templates: foundations, design landscape and research directions. [online], 2016, [2019-12-09]. Available from: http://www.resnovae.org.uk/fccsuclacuk/images/article/sct2016.pdf

[16] Sherman, L. Bitcoin's Energy Consumption Can Power An Entire Country – But EOS Is Trying To Fix That. [online], 2018, [2019-12-09]. Available from: https://www.forbes.com/sites/shermanlee/2018/04/19/bitcoinsenergy-consumption-can-power-an-entire-country-but-eos-istrying-to-fix-that/#7a45bd5b1bc8

[17] Wahab, A.; Memood, W. Survey of Consensus Protocols. [online], 2018, [2019-12-09]. Available from: https://arxiv.org/ftp/arxiv/papers/1810/1810.03357.pdf

[18] IOHK Limited. About Input Output - IOHK. [online], [2019-12-09]. Available from: https://iohk.io/en/about/

[19] Chan, J. Who is Cardano Founder Charles Hoskinson? [online], 2019, [2019-12-09]. Available from: https://www.asiacryptotoday.com/who-is-cardano-founder-charles-hoskinson

[20] Cardano Foundation. What is Cardano? [online], [2019-12-09]. Available from: https://www.cardano.org/en/what-is-cardano/

[21] Kiayias, A.; Russell, A.; David, B.; et al. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. [online], 2017, [2019-12-09]. Available from: https://eprint.iacr.org/2016/889.pdf

[22] IOHK Limited. The Plutus Language Implementation and Tools. [online], [2019-12-21]. Available from: https://github.com/input-output-hk/plutus

[23] IOHK Limited. Plutus Playground. [online], [2019-12-09]. Available from: https://prod.playground.plutus.iohkdev.io/

[24] Joeris, B. Haskell Fundamentals Part 1. [online], 2013, [2019-12-09]. Available from: https://app.pluralsight.com/library/courses/haskell-fundamentals-part1/

[25] Taylor, N. Functional Programming: The Big Picture. [online], 2018, [2019-12-09]. Available from: https://app.pluralsight.com/library/courses/functional-programming-big-picture/

[26] Joeris, B. Haskell Fundamentals Part 2. [online], 2014, [2019-12-09]. Available from: https://app.pluralsight.com/library/courses/haskell-fundamentals-part2/

[27] IOHK Limited. Cardano Roadmap. [online], [2019-12-16]. Available from: https://cardanoroadmap.com/en/

[28] Vejrážková, Z. *Business Process Modeling and Simulation: DEMO, BORM and BPMN*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2013.

[29] Dietz, J. L. G. *Enterprise Ontology: Theory and Methodology*. Berlin: Springer, 2006, ISBN 978-3540291695.

[30] Enterpirse Enigineering Institute. Enterpirse Enigineering and DEMO. [online], [2019-12-09]. Available from: http://www.ee-institute.org/en/demo

[31] Phalp, K. T. The CAP Framework for Business Process Modelling. *Information and Software Technology*, volume 40, no. 13, 1998: pp. 731–744.

[32] Dietz, J. L. G. The Essence of Organisation - Preview. [online], 2015, [2019-12-09]. Available from: http://www.ee-institute.org/download.php?id=130&type=doc

[33] Skotnica, M.; van Kervel, S. J.; Pergl, R. A DEMO Machine - A Formal Foundation for Execution of DEMO Models. [online], 2017, [2019-12-16]. Available from: https://link.springer.com/content/pdf/10.1007%2F978-3-319-57955-9_2.pdf

[34] Dietz, J. L. G. DEMO Specification Language 3.6. [online], 2017, [2019-12-09]. Available from: http://www.ee-institute.org/download.php?id=208&type=doc

[35] Hornáčková, B.; Skotnica, M.; Pergl, R. Exploring a Role of Blockchain Smart Contracts in Enterprise Engineering. [online], 2019, [2019-12-16]. Available from: https://link.springer.com/content/pdf/10.1007%2F978-3-030-06097-8_7.pdf

[36] Pintado, O. L.; Bañuelos, L. G.; Dumas, M.; et al. CATERPILLAR: A Business Process Execution Engine on the Ethereum Blockchain. [online], 2018, [2019-12-10]. Available from: https://www.researchgate.net/publication/326988238_CATERPILLAR_A_Business_Process_Execution_Engine_on_the_Ethereum_Blockchain

[37] Skotnica, M. Lecture notes in Software Team Project: Functional Specification - PSI Contract Designer. 2019.

[38] Microsoft Corporation. .NET Website. [online], [2020-01-07]. Available from: https://dotnet.microsoft.com/

[39] Shopify. Shopify: Liquid Reference. [online], [2020-01-07]. Available from: https://help.shopify.com/en/themes/liquid

[40] Ros, S. Fluid Template Engine. [online], [2020-01-07]. Available from: https://github.com/sebastienros/fluid

[41] Mirkovic, J. Blockchain Cook County — Distributed Ledgers for Land Records. [online], 2017, [2020-01-07]. Available from: https://illinoisblockchain.tech/blockchain-cook-county-final-report-1f56ab3bf89

[42] Farlex Incorporated. Legal Dictionary. [online], [2020-01-07]. Available from: https://legal-dictionary.thefreedictionary.com/

Appendix A

Acronyms

AM Action Model

BPM Business Process Modeling

BPMN Business Process Model and Notation

CM Construction Model

DEMO Design and Engineering Methodology for Organizations

FM Fact Model

IOHK Input Output HK Limited

OCD Organization Construction Diagram

OFD Object Fact Diagram

PM Process Model

PoS Proof of Stake

PoW Proof of Work

PSD Process Structure Diagram

TPD Transaction Pattern Diagram

UTXO Unspent Transaction Output

Appendix B

Contents of enclosed CD

```
readme.txt ............ the file with CD contents description
src ....................... the directory of source codes
  scgenerator ................... 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
```