FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Generation of Plutus Smart Contracts from DasContract models
Student:	Bc. Martin Drozdík
Supervisor:	Ing. Marek Skotnica
Study program:	Informatics
Branch / specialization:	Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

DasContract is an open-source Smart Contract format aiming to provide a visual language for designing legal contracts between parties that can be used to generate executable blockchain Smart Contracts. DasContract is currently capable of generating Solidity contracts running on the Eterium blockchain.
This thesis investigates the possibilities of generating Plutus Smart Contracts from the DasContract model running on the Cardano blockchain.

Steps to take:
- Review the Cardano blockchain, Haskell, and Plutus languages
- Design and create a proof-of-concept generator from DasContract models to Plutus blockchain language
- Summarize the state of the proof-of-concept generator and propose the next steps in implementing further features

Electronically approved by Ing. Michal Valenta, Ph.D. on 25 November 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Generation of Plutus Smart Contracts from DasContract models

Bc. Martin Drozdík

Faculty of Information Technology
Supervisor: Ing. Marek Skotnica

April 19, 2022

Acknowledgements

Thank you to my supervisor, Ing. Marek Skotnica, for providing guidance and feedback throughout this thesis.

Declaration

In Prague on April 19, 2022 . . . .

Citation of this thesis

Drozdík, Martin. *Generation of Plutus Smart Contracts from DasContract models.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstract

This thesis explores the Cardano blockchain, its native functional language Plutus, and the possibilities to generate Cardano smart contracts from the semi-visual smart contract modeling language DasContract.

The Cardano smart contract generator has the potential to generate decentralized, autonomous, and secure electronic contracts. Users may stop caring about boilerplate code or complex insights of the Plutus language and start developing robust, less error-prone, and less ambiguous smart contracts. Such contracts may remove the need for central authorities and positively impact our juridical system and society.

In this thesis, a proof-of-concept implementation of the Plutus smart contract generator has been analyzed, designed, implemented, and tested. A case study of a funds locking contract has been created to demonstrate the capabilities of the Plutus contract generator. The generators' source code is publicly available to aid further research. This thesis summarized implemented features of the Plutus generator, highlighted problematic areas, and proposed additional features to be studied.

Keywords Proof of concept Plutus generator, DasContract generator, Cardano, Plutus programming language, Plutus contracts, blockchain, smart contracts

Abstrakt

Tato práce zkoumá Cardano blockchain, jeho programovací jazyk Plutus a možnosti generování chytrých kontraktů na Cardano blockchainu z částečně vizuálního modelovacího jazyka DasContract.

Generátor chytrých smluv, běžící na blockchainu Cardano, má potenciál tvořit decentralizované, autonomní, bezpečné a elektronické kontrakty. Uživatelé se již nebudou muset zabývat nezajímavým kódem a složitostmi jazyka Plutus. Stačí se zaměřit na tvorbu robustních, nechybových a zřetelně fungujících chytrých kontraktů. Chytré kontrakty dělají z centrálních autorit zbytečné prostředníky a pozitivně ovlivňují právní systémy i naši společnost jako takovou.

V této práci je implementován prototyp generátorů chytrých kontraktů v jazyce Plutus. Prototyp prošel analýzou, návrhem, implementací a otestováním. Zároveň byl navržen a vytvořen konkrétní příklad chytrého kontraktu, schopného na nějaký čas zmrazit finanční prostředky, jako demonstrace schopností Plutus generátoru. Zdrojový kód generátoru je veřejně dostupný jako open source pro podporu dalšího výzkumu. Tato práce nakonec shrnula, co Plutus generátor umí, na jaké problémy se narazilo, a poskytuje návrhy na další funkčnosti pro studii a výzkum.

Klíčová slova Prototyp Plutus generátoru, DasContract generátor, Cardano, programovací jazyk Plutus, Plutus smlouvy, blockchain, chytré kontrakty

Contents

List of Figures

List of Tables

List of Listings

Introduction

DasContract is a visual language capable of defining smart contracts [1]. This thesis aims to create a Cardano smart contract generator from DasContract files.

Motivation

Current society closes deals and arrangements using contracts. These contracts commonly take the form of signed papers with clearly stated conditions of the agreement. Suppose an interested party breaks the conditions of a paper contract. In that case, other parties must rely on a juridical system to enforce the conditions of the contract, which can take years and cost a significant amount of resources [2].

Blockchain smart contracts are a form of secure code that has the potential to replace standard paper contracts and eliminate the need for central authorities because it is capable of enforcing certain conditions on its own [1][2].

Creating smart contracts is a non-trivial task requiring programming skills, experience, and knowledge, especially for contracts handling a currency. These contracts must behave as intended and be secure against attacks. DasContract is a tool aiming to provide a better and more reliable way to design smart contracts using visual languages, such as Business Process Model and Notation (BPMN) or Decision Model and Notation (DMN).

The DasContract editor currently supports generating contracts for the Ethereum blockchain. However, there are more blockchain networks capable of running smart contracts. One of them is the Cardano blockchain. If a new generator capable of creating Cardano contracts from the DasContract format existed, it would empower DasContract as a potent abstract format for designing smart contracts. The reason is that Cardano is fundamentally different from the Ethereum blockchain in many areas. It would also provide a new way of creating Cardano smart contracts, which may be beneficial to the Cardano comunity since it is a new and still developing blockchain technology.

Problem statements

Currently, there is no way to generate Cardano smart contracts based on the DasContract format. This thesis aims to create a proof of concept generator of Cardano smart contracts with DasContract file as an input.

Ethereum and Cardano blockchains are in many fundamental properties different. Creating a new DasContract generator to Cardanos' language Plutus will deepen research of the DasContract format and potentially discover abstraction flaws with the DasContract format itself.

A new tool such as DasContract to Plutus convertor could positively impact the Cardano developer community since Cardano documentation and instruments are scarce. Cardano is a developing platform that could use any tools it can get.

Objectives

The main objective of this thesis is to investigate the possibilities of the Cardano technology, analyze, design, and implement an open-source contract generator based on the DasContract format.

The output of the smart contract generator must be an executable Haskell file representing the logic described by a DasContract file. Further research of the generator or DasContract should be suggested if required.

The state of the smart contract generator must be summarized with supported and unsupported features, along with recommended features for future development and research.

The objectives of this thesis do not include developing the DasContract language or any other DasContract-related products. Furthermore, objectives do not include any development of a blockchain wallet or a User Interface (UI).

Structure and methodology

When approaching the goals of this thesis, the Cardano blockchain has been thoroughly investigated, and the Plutus language has been carefully learned. Along with learning Plutus, the newest version of DasContract has been researched and consulted with its authors and maintainers.

After all knowledge has been accumulated, the practical part, according to the goals, has been implemented. First, a prototype contract has been constructed; then, iteratively, the proof-of-concept generator has been designed and implemented.

The thesis has been finished by reporting the resulting product and recommending future research.

This thesis is organized as follows:

- In chapter 1, the Cardano blockchain is revied along with blockchain basics and the Plutus development language.
- In chapter 2, the DasContract format and its inner workings are described.
- In chapter 3, the proof-of-concept Plutus smart generator is described along with fundamental implementation details.
- In chapter 4, a case study of a funds locking smart contract is described.
- In chapter 5, the state of the Plutus smart generator is summarized with supported, unsupported, and future-recommended features.

Chapter 1

Review of the Cardano blockchain

"Cardano is a proof-of-stake blockchain platform: the first to be founded on peer-reviewed research and developed through evidence-based methods. It combines pioneering technologies to provide unparalleled security and sustainability to decentralized applications, systems, and societies."

"With a leading team of engineers, Cardano exists to redistribute power from unaccountable structures to the margins – to individuals – and be an enabling force for positive change and progress." [3]

1.1 Relevant blockchain cryptography

Blockchain technologies heavily utilize cryptography. Two primary tools are asymmetric cryptography and hash functions.

Asymmetric cryptography is based on having two keys – private key and public key. Public keys are distributed amongst users using a distribution system. Private keys are the secrets of each owner/source [4].

One major use case of asymmetric cryptography is signing. If something is signed, it can be easily verified where the data came from, its origin can not be disputed, and that the data integrity holds. Private keys are used to encrypt (=sign) data. They can only be decrypted using the respective public key. This enables everyone on the network to verify data belonging to a user since he is the only one that could have encrypted it (using his private key) [4].

$$data = D_{pubkey}(E_{privkey}(data))$$

A **hash function** is a one-way function from set X to set Y (hash). Usually, elements of set X have arbitrary sizes, and elements of set Y have a fixed size. Hash functions are deterministic, and the map from set X to set Y should appear as random and evenly distributed. It also should not be

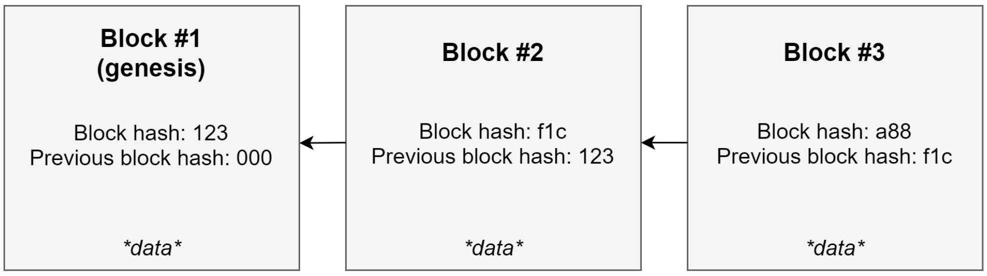


Figure 1.1: A possible structure of a blockchain network

possible to figure out items mapping from set Y to a specific item in set X in a reasonable amount of time – making it one-way [5].

Hashing has many practical use cases. For example, a hash of a data string should always be the same; thus, rehashing data can serve as an integrity check when downloading large or critical data. Another use case is the commit scheme, where commitment is done using a data hash instead of the original data since the original data should be kept secret. Commitment is then proven with the original data when it is no longer needed to keep it a secret – it is borderline impossible to figure out the original data just from the hash [6][5].

1.2 Blockchain technology

Blockchain technology introduced by Satoshi Nakamoto is a decentralized, distributed and autonomous peer-to-peer network. The network is trustless, meaning there must exist some consensus mechanism on which the network users agree [1][7].

Blockchain data is contained in blocks, abstractly connected by a chain – hence the name blockchain. The blocks can be logically bound by hashes of themselves and their predecessors. Changes in a block in the middle of a chain would require rehashing of all upcoming blocks, which is an enormous obstacle and ensures chain immutability. However, blocks of data, sometimes named transactions, behave and are built differently on various blockchains [7][8][9].

A significant property of blockchain networks is their immutability, which ensures security against attacks and forgeries. Every transaction is kept, and every value or state can be backtracked and recalculated [1][7].

Blockchain technology represents a major resource for various applications, such as decentralized finance, asset exchange, decentralized applications, or even decentralized legal contracts. Possibilities are many.

Blockchain is mainly known for its use with Bitcoin as its underlying technology. Since Bitcoin, many more blockchains have emerged, such as Ethereum or Cardano. These blockchains do follow the ideology on which Bitcoin is built but offer so much more than just a decentralized currency [1][8][9].

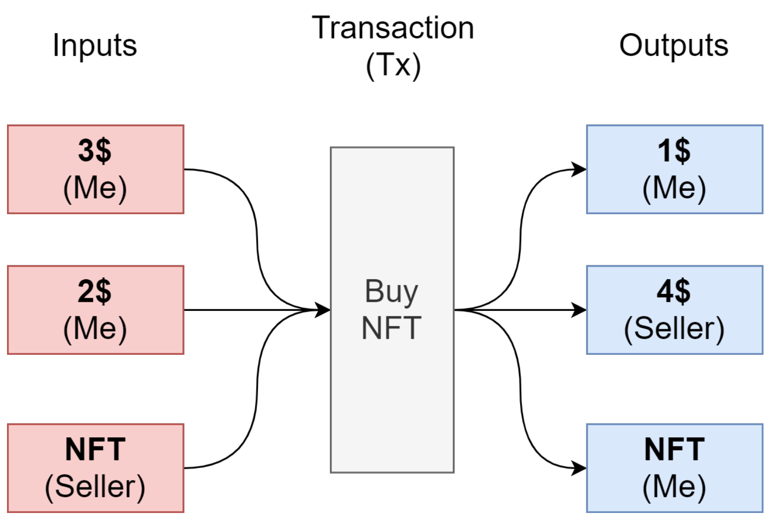


Figure 1.2: A simplified example of UTXO

1.3 UTXO

Unspent Transaction Output (UTXO) is an accounting model enabling keeping track of every wallets' state. UTXO can be imagined as three working parts: transactions, inputs of these transactions, and outputs of these transactions. Instead of keeping track of wallets' state by centralized ledgers (similar to banks), the UTXO system keeps track of wallets' states by summing up unspent outputs assigned to the wallet. These unspent outputs can be spent by a transaction, which will produce other unspent outputs, redistributed accordingly [10].

A real-world analogy would be imagining unspent outputs as banknotes and coins in my wallet. I have five dollars consisting of a two-dollar bill and a three-dollar bill. That means there are two unspent outputs, each having its respective value. I can go to a shop and buy a Non-Fungible Token (NFT) for four dollars. The *"buy"* operation is a transaction. To be able to purchase the NFT, I need at least four dollars. I am in luck; I have five dollars total. I can use these dollars (unspent outputs) as inputs to the *"buy"* transaction. The transaction results in two new unspent outputs. One unspent output has a value of four dollars and goes to the seller. The second unspent output has a value of one dollar and goes back to me because I needed to spend only four dollars but got banknotes for five dollars minimum.

The example is visualized at figure number 1.2. The NFT is also an input and an output of the transaction, where the owner changes from the seller to me.

A fundamental rule is that the number of resources consumed by the transaction (inputs) equals the transactions' output. There are exceptions to this rule, such as fees, the genesis block, or token minting/burning [10][11].

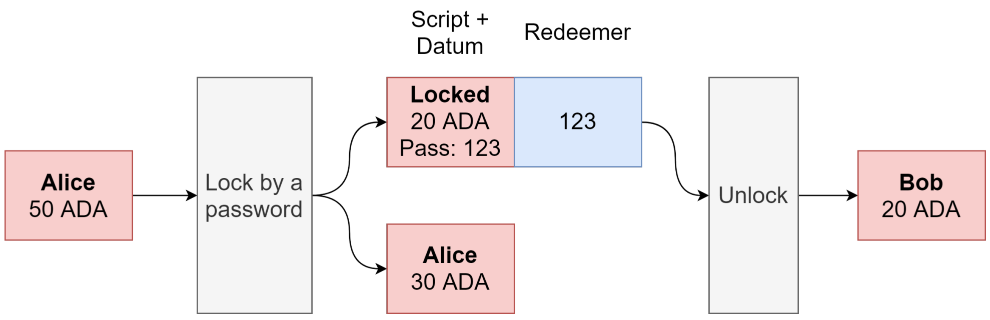


Figure 1.3: A simplified example of EUTXO

Another fundamental rule is that spent outputs can not be spent again or altered [11].

The address of an unspent output determines who can spend it (who owns it). If a transaction spending these outputs appears on the blockchain, their owner must cryptographically sign it using their private key. This ensures that no one can spend somebody else's outputs [11].

The Bitcoin blockchain uses the UTXO model, just as described in this section [10].

1.4 EUTXO

Extended Unspent Transaction Output (EUTXO) is an extension of the UTXO model. UTXO validates output owners using a cryptographic signature. EUTXO extends the behavior of inputs and outputs and enables using custom logic, not only wallet signatures [10][11].

For example, an unspent output can be created and programmed to be claimed by anyone that knows the password. Another example would be an unspent output that a person can claim after a deadline has passed.

Every unspent output can be equipped with a datum – an arbitrary data value and a script – a piece of arbitrary logic. When a transaction tries to spend the output, it does so with a redeemer – also arbitrary data. The script (also called a validator script) checks the arbitrary logic when an output tries to be redeemed with a redeemer [10][11].

The output needs to contain only the datum hash, but the redeeming transaction must send a correct datum value. The script has its address and can lock various valuables, including funds. The validator script accepts a datum, a redeemer, and a context when it is run [10][11].

The EUTXO is **Turing-complete**, thus powerful enough to implement even complex smart contracts [10].

1.5 Haskell

"Haskell is an advanced purely-functional programming language. An open-source product of more than twenty years of cutting-edge research, it allows rapid development of robust, concise, correct software. With strong support for integration with other languages, built-in concurrency and parallelism, debuggers, profilers, rich libraries and an active community, Haskell makes it easier to produce flexible, maintainable, high-quality software." [12]

Haskell is a **functional**, **declarative** language. In contrast, imperative languages instruct the machine on how an operation should be executed – some languages are more specific than others. Declarative functional languages declare what should be the result. The compiler or some other tools then infer the specific execution steps [13].

A key feature of Haskell is the **lack of side effects**. A function always returns the same result for the same input – there are no global variables, memory access, etc. The only tool with side effects is the IO monad, which ensures communication with the "outside world" such as file Input/Output (I/O) operations, standard output communication, etc. [14][15]

Haskell is a **statically typed** language. The compilator checks if all types are in order and only then compiles the program. The compiler is also capable of inferring types from context [14].

Haskell supports a **packaging system**, where pieces of code can be separated into their own respective packages and then referenced when needed. This promotes a good program structure and enables easy code sharing [14].

Haskell supports **lazy evaluation**. Expressions are not evaluated until they are needed. Since Haskell is a purely-functional language, this property should be hidden to the programmer [14].

1.6 Cardano

"Cardano is an open source proof-of-stake blockchain project that began in 2015 to address existing blockchain challenges in the design and development of cryptocurrencies. It aims to provide a more balanced and sustainable ecosystem that better accounts for the needs of its users as well as other systems seeking integration." [16]

The main driving force behind Cardano is the IOHK company, founded in 2015 by Charles Hoskinson and Jeremy Wood [17].

Cardanos' accounting model uses the EUTXO. It empowers Cardano to support monetary operations with its native currency ADA, minted tokens, or even complex smart contracts, capable of executing complex logic, thanks to being Turing-complete [8].

1.6.1 The consensus algorithm

Cardano uses a proof-of-stake algorithm as its consensus mechanism called Ouroboros. Ouroboros is *"a proof-of-stake protocol that provides and improves the security guarantees of proof-of-work at a fraction of the energy cost. Ouroboros applies cryptography, combinatorics, and mathematical game theory to guarantee the protocols' integrity, longevity, and performance, and that of the distributed networks that depend upon it."* [18].

The Ouroboros protocol perceives time in epochs, which consists of time slots. One epoch currently stretches over five days with exactly 432.000 time slots. However, these time intervals are not guaranteed to persist, and the real-time length of epochs and timeslots may change in the future [19][20].

Each time slot may elect a slot leader, which mines the next single block. Slot leaders are stake pools, with a certain amount of stake entrusted from the blockchains' users. The more stake the pool controls, the higher probability of being selected as the slot leader. Successfully mining the next block rewards the stake pool and its staking participants (depending on the stake pools settings) [18][19][20].

Notable is the power efficiency compared to the proof-of-work consensus algorithms, such as in the Bitcoin blockchain. Secure and reliable proof-of-stake algorithms are considerably more complex than proof-of-work algorithms; however, their efficiency in terms of power consumption is superior many times over [18].

1.6.2 Native tokens

Cardanos' native currency is ADA. One ADA is further divisible to one million Lovelace. ADA is the currency used to pay transactions fees. Its supply is finite. The initial ADA coins are minted in the genesis block and can not be minted again [21][22].

Cardano natively supports work with any type of tokens. Sending and working with ADA is equally similar to working with other tokens [21][22].

User-defined tokens can be minted (=created) using a minting script attached to a EUTXO. The script defines the minting behavior. For example, one token can be minted for 1 ADA. Or, the token can be minted only once [22].

One categorization for tokens is fungibility. Fungible Token (FT) is interchangeable. For example, it does not matter if I have this or that ADA token – one ADA is still one ADA; it does not matter which one I have. Non-Fungible Token (NFT) is always unique and one of a kind (not interchangeable). NFTs can be used to identify an art piece or an ongoing smart contract process [21][22].

There are several ways of burning (=destroying) tokens. One is sending the token to a void wallet of which nobody knows the private key – thus unable to send the tokens elsewhere. Another way is creating a EUTXO with a script that always fails – thus money in the EUTXO can not be further processed by a transaction [22].

1.6.3 Plutus

Plutus is a language capable of developing Cardano smart contracts. Plutus is built on the Haskell programming language, and Plutus programs are essentially Haskell programs [23].

Plutus is used to develop the off-chain and on-chain parts of the smart contract. The off-chain code is a code that runs on the clients' side, possibly in his preferred wallet application. The on-chain code is a code that runs on the Cardano blockchain [10][23].

The off-chain code is responsible for creating transactions and submitting them to the blockchain. Besides that, it can do many more operations, such as client-side validation. The off-chain code is generally built as a Contract monad, which offers one or more endpoints for a user to invoke [10][23].

The on-chain code is exclusively for validation scripts and minting policies. By the nature of the blockchain, this code is run only on-demand, thus being unable to establish communication with a user on its own – for example, an event notification [10][23].

Both on-chain and off-chain are written concurrently to eliminate redundancies and simplify the code, and both are written in Haskell. Later, the on-chain part is compiled into Plutus Core – code running on Cardano – and sent to the blockchain. Since the source of the on-chain code is written in Haskell, it can be used, for example, by the off-chain part to validate the transaction even before it reaches the blockchain entirely. Plutus is built to be fully deterministic, and everything that is successfully validated off-chain will be successfully validated on-chain. One single exception is user concurrency – for example, two users can send the same transaction, and only one will get validated because duplicities are not allowed [10][23][24][25].

By Haskells' nature, the Cardano blockchains' design, and Plutus' Core design, Plutus is a secure and reliable way to develop smart contracts. The correct output of a function is more easily provable and verifiable, which effectively eliminates any problems that other blockchains have with their imperative-oriented languages [26]. Determinism and predictability are one of the primary focuses of the Plutus platform. They significantly promote the safety of the blockchain and user experience [10][24][25].

1.7 Summary

Blockchain is a decentralized peer-to-peer network. The backbones of these networks are immutable append-only chains of transactions. The chains' immutability and user identification are ensured through cryptography.

One blockchain technique to store and organize transactions is the UTXO, where tokens are handled like traditional coins in a wallet – not in centralized ledgers. The total value of someones' tokens is the sum of their unspent outputs.

Cardano is an advanced proof-of-stake blockchain with the support of smart contracts. Cardano uses its own unique EUTXO model, where unspent outputs can contain arbitrary logic – smart contracts. Cardano natively supports work with its currency ADA and user-defined tokens – working with both is essentially the same.

Cardano smart contracts can be developed using the Plutus programming language. Plutus programming is done in Haskell. Plutus programs consist of off-chain and on-chain code. The off-chain primarily builds and submits transactions to the blockchain. The on-chain code is compiled, sent, and works on the blockchain as validation. A key advantage of Plutus contracts is that the on-chain code can be also utilized in the off-chain part. It removes redundancy and adds to the deterministic nature of Plutus programming.

Chapter **2**

Review of the DasContract format

DasContract is currently a semi-visual language, aiming to provide a platform and a format for an efficient and less error-prone way to design and deploy smart contracts. Smart contracts are a way of eliminating the current form of legal contracts, which contain ambiguities. Furthermore, it may take longer for existing contracts to wait before the legal system sorts out any non-compliant parties or individuals [1]. Table 2.1 describes the contract maturity model and contracts' ambiguities, risk, or error-proneness.

The authors of the DasContract propose three key components to solve contract issues in the current form:

Human Understanding *part defines a contract between multiple parties*

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

Table 2.1: Table of contract maturities [1]

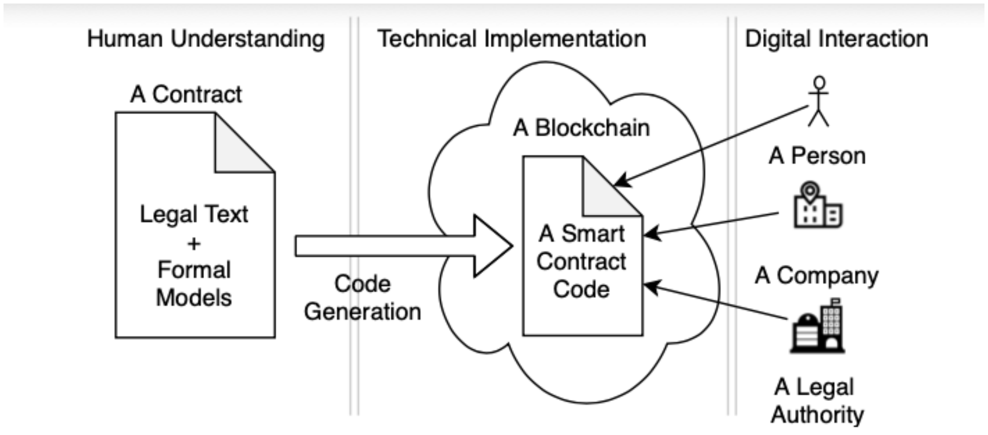


Figure 2.1: High-level abstraction of how DasContract should work [1]

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

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

Digital Interaction *is a part where people, companies and legal authorities can interact with the agreed upon contracts. Since the contract is in a blockchain, the interaction is fully digital, and thanks to cryptography can also be legally binding. Blockchain by design also provides an audit trail of all actions performed by the parties and ensures that the agreed upon contract is executed correctly.* [1]

This thesis works with the presently most recent version DasContract 2.0. DasContract 2.0 format perceives a smart contract as a combination of three key areas – **contract process**, **contract data model**, and **contract users**. Furthermore, these areas consist of several editors and languages working together towards fully defining all essential logic and components of a smart contract. These editors and languages include [1][27]:

- BPMN editor for processes
- Decision Model and Notation (DMN) editor for business tasks
- User forms editor for user activities
- Data model editor
- Users and roles editor
- Blockchain-specific code

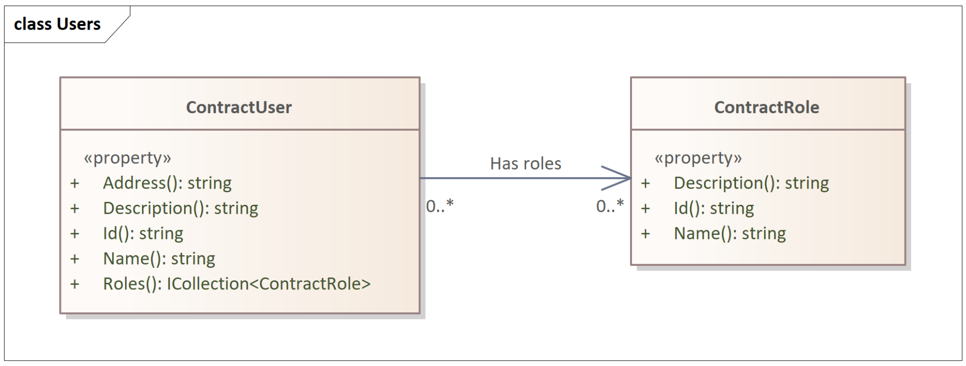


Figure 2.2: High-level abstraction of how DasContract user models work

DasContract **editors** create and edit DasContract files and output a DasContrac file with the extension `.dascontract`. This file contains all essential data for generating the final smart contract [27].

A **smart contract generator** consumes DasContract files and generates respective blockchain-specific code. At this time, there is a generator capable of transforming DasContract files into Solidity contracts running on the Ethereum blockchain network [28]. This thesis aims to add another generator capable of making Plutus smart contracts for the Cardano blockchain network.

Once the final smart contract is generated, it can be **deployed** on the blockchain network. Interaction with the contract can be done multiple ways, depending on the blockchain; however, a wallet or other off-chain program should be capable of providing a UI-friendly interface to cater to people with average blockchain knowledge.

2.1 User definition

The current DasContract 2.0 looks at users as a collection of individuals. Each individual (user) may perform various roles [27].

Roles have a simple name and a symbolic description [27].

Users have a name, symbolic description, roles, and, most importantly, a public key of their wallet [27].

Users or roles can be later utilized throughout the contract as an identification. One of the essential use cases for users and roles is the assignment to a user activity – only certain users, identified directly or indirectly by their roles, may submit and complete a user activity [27].

2.2 Data model definition

DasContract provides the possibility to design a custom data model. The data model, if possible, is used to generate its image in the resulting blockchain smart contract. Its usage may be arbitrary and depends on the contracts' developer. The data model image may look different depending on the blockchains' language [27][28].

The DasContract data model consists of:

- Enums
- Tokens
- Entities

Enums are a trivial structure of a finite set of values. An entity property may reference these values [27].

A token is a blockchain token with important specified properties, such as a symbol, fungibility, source mint script identification, and others. Unfortunately, this structure is heavily aimed at the Ethereum blockchain and the ERC-20 standard [27]. Blockchains such as Cardano treat tokens differently and do not require attributes like a transfer script or if it was issued, which the DasContract token defines [22][21].

Entities are a set of attributes and properties. Entity attributes include entity name, entity identification and check, and if the entity is a root entity. The root entity marks the entity as the entry point for data model generation. There must be exactly one root entity. Entity properties (members) also contain a set of attributes. Essential property attributes are identification and display name. Most importantly, the property data type for the blockchains' language must be derivable; therefore, further data type attributes must be defined. The user can specify if a property is mandatory and property type must be selected [27][29]:

- Property type set to "single" means a single value property with a specified data type.
- Property type set to "collection" means a collection of values of a specified data type. The data collection structure is unspecified but expected to be an integer-indexable list-style collection.
- Property type set to "dictionary" means a dictionary (key-value) collection of specified key type and data type.

Property types may be Int, `Uint`, `Bool`, `String`, `DateTime`, `AddressPayable`, `Address`, `Reference`, or `Enum`. `Reference` properties contain another instance of an entity. The property then requires another attribute for specifying the target entity identification [27].

Listing 2.1 Example of a DasContract data model XML definition

```xml
<DataTypes>
  <Entity Id="Root" Name="Datum" IsRootEntity="true">
    <Property Id="iNumber" Name="interestingNumber"
      IsMandatory="true"
      PropertyType="Single" DataType="Int" />
    <Property Id="iMessages"  Name="interestingMessages"
      IsMandatory="false"
      PropertyType="Collection" DataType="String" />
    <Property Id="iEntity" Name="interestingEntity"
      IsMandatory="true"
      PropertyType="Single" DataType="Reference"
      ReferencedDataType="SecondEntity" />
    <Property Id="iEnum" Name="interestingEnum"
      IsMandatory="true"
      PropertyType="Dictionary" KeyType="Int"
      DataType="Enum" ReferencedDataType="Enum1" />
  </Entity>

  <Entity Id="SecondEntity" Name="SecondEntity"
    IsRootEntity="false" />

  <Enum Id="Enum1" Name="Enum1">
    <Value>Value1</Value>
    <Value>Value2</Value>
    <Value>Value3</Value>
  </Enum>
</DataTypes>
```

2.3 Process definition

The process definition is a subset of the BPMN diagram stating how the resulting smart contract should behave [27].

"The Business Process Model and Notation (BPMN) specification provides a graphical notation for specifying business processes in a Business Process Diagram. Its goal is to support Business Process Modeling by providing a standard notation that is comprehensible to business users yet represents complex process semantics for technical users.

Business Process Modeling Notation has become the de-facto standard for business processes diagrams. It is intended to be used directly by the stakeholders who design, manage and realize business processes, but at the same time be precise enough to allow BPMN diagrams to be translated into software *process components. BPMN has an easy-to-use flowchart-like notation that's independent of any particular implementation environment."* [30]

The currently used subset of the BPMN notation includes the start event, end event, exclusive gateway, parallel gateway, sequential multi-instance, parallel multi-instance, user task, script task, business rule task, call activity, and the timer boundary event [27].

As is evident with the inclusion of the call activity, the DasContract process can contain multiple subprocesses. There must be precisely one process marked as executable, marking the entry point for smart contract generators [27].

The **start event** is the initial entry point for a process. If the process is executable, the entire contracts' entry point is the start event [27].

The **end event** either transitions to the caller process or ends the whole contract if there is no caller process [27].

The **exclusive gateway** chooses only one possible output connection. Output connections have conditions, where only the one selected output connection should be truthful [27].

The **parallel gateway** simultaneously proceeds to all output connections [27].

Any task can be a **sequential multi-instance**. The task is then executed in a loop. The number of loops is determined by loop cardinality. If a loop collection property is set, the number of loops is the length of the collection, and each loop will provide the nth element of the collection (foreach loop) [27].

Any task can be a **parallel multi-instance**. The task is then executed multiple times, simultaneously. The number of instances is determined by loop cardinality. If a loop collection property is set, the number of instances is the length of the collection, and each instance will provide the nth element of the collection (parallel foreach) [27].

User tasks represent a task with required input from a user – a form. The form is defined using a special XML notation developed by Bc. Petr Ančinec, in his masters' thesis, Domain-Specific Languages for Off-chain UI in Decentralized Applications. The notation contains data-binding syntax to store values in the data model directly and effortlessly [31]. Additionally, user tasks provide a space for a validation script, where the form can be manually validated. This script may also save or further process the data model, or other operations. User tasks may select an assignee, candidate users, or candidate roles. These are the contracts' users and roles. They must be the only users capable of completing the task. The proof of completion is usually a digital signature with respective private keys [27][29].

Script tasks run automatically and provide a place to code arbitrary contract logic. The capabilities of the blockchain limit the logic [27][29].

Business rule tasks use DMN notation to define rules and decitions [27].

"DMN is a modeling language and notation for the precise specification of business decisions and business rules. DMN is easily readable by the different

types of people involved in decision management. These include: business people who specify the rules and monitor their application; business analysts.

DMN is designed to work alongside BPMN and/or CMMN, providing a mechanism to model the decision-making associated with processes...” [32]

The DMN to Ethereum generation is currently under research and development [29].

Call activities invoke a subprocess. Recursive and multi-level subprocess calls are allowed. The subprocess starts at the start event and returns when an end event is encountered [27][29].

The **timer boundary event** may be placed on any task. When the boundary event timer times out, the process flow continues on the marked boundary events’ timed out path. The timeout can be set using a specific timeout date or a duration for which the task can be uncompleted [27].

2.4 Evaluation tools

The contracts’ process must be appropriately modeled and evaluated in the target blockchain language. There is no fixed guideline since different blockchains and respective languages vary. However, abstract tools generally describe the behavior and are universally implementable in Turing-complete languages.

The **deterministic finite state machine** is formally defined as a tuple of five elements $(S, \Sigma, \delta, q_0, F)$ where [33]:

- S is a finite set of states
- q_0 is the initial state
- δ is a transition function $\delta : S \times \Sigma \to S$
- F is a set of final states where $F \subseteq S$
- Σ is an input alphabet (a finite non-empty set)

The deterministic state machine is a suitable code design base for a program to emulate a DasContract process. The DasContract process begins with the start event (q_0) then proceeds to transition (δ) depending on values in the data model or simply to the next available state. The process finishes in an end event (F).

It must be noted that a deterministic state machine cannot fully simulate the DasContract process. However, it is a programmatically solid baseline for the code design, which requires a small number of modifications. For example, the deterministic state machine can not suitably handle multiple processes, working with the persistent data model or concurrent tasks.

The **call stack** structure may be utilized to support multiple processes and call activities [34]. Stack is a Last In – First Out (LIFO) structure. The

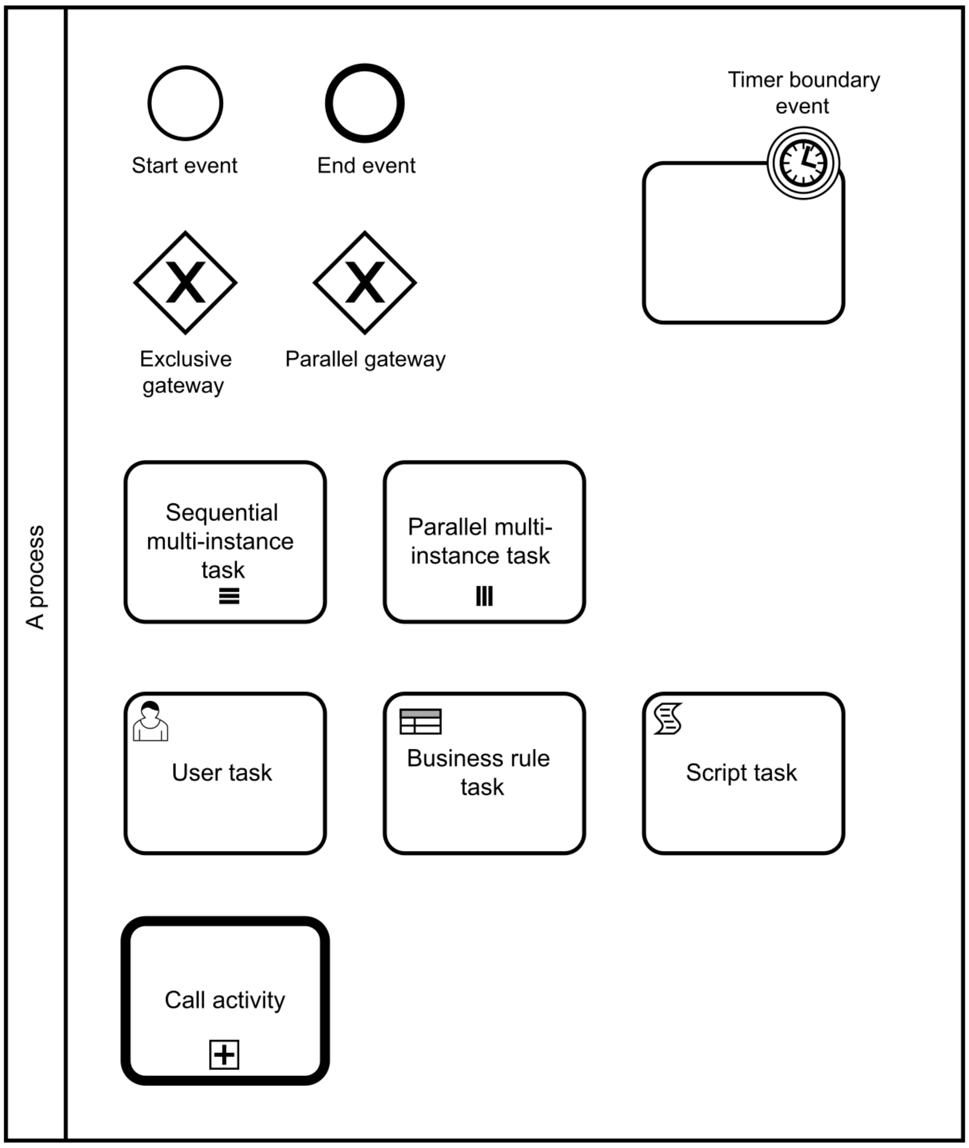


Figure 2.3: The BPMN subset used in the DasContract format

initial call stack is empty. When a call activity is encountered, a returning state is pushed in the call stack, and the process referred by the called activity is invoked – the current state is now at the invoked processes' start event. The same happens recursively, enabling call activities within call activities, even recursive subprocesses. When an end event is encountered, a state is popped from the call stack, and the contract continues from the popped state. If there is no state to pop, the end events mean the end of the entire contract.

A state machine, a call stack, both with access to the persistent data model and user information, is a suitable code design basis for any blockchain language.

2.5 Summary

The DasContract format is trying to provide a solid platform for smart contracts. Using DasContract should result in fewer errors, fewer ambiguities, and a better understanding of the contracts' logic.

DasContracts are made in editors, where users can design their contracts. Afterward, the DasContract files can serve as an input to a blockchain generator. The targeted blockchain depends on the generator. Generated code can then be executed on the blockchain network.

The DasContract format consists of three subformats: users, data model, and processes. Users define the roles and specific personas interacting with the contract. The data model serves the programmer for arbitrary usage. The process, defined using a subset of the BPMN language, describes the business logic of the contract.

The DasContract format eventually has to be translated into a blockchain code. A good and solid code design base is to evaluate processes using a state-machine-like execution with the support of a call stack for calling subprocesses and access to the persistent data model. Additional details need to be figured out based on the specific blockchain.

Chapter 3

Proof-of-concept Plutus generator

The DasContract format currently supports the generation of Solidity smart contracts for the Ethereum blockchain network [27]. A new generator into the Plutus language would expand DasContract possibilities, test the correctness of the DasContract format, and solidify its universality and blockchain-language independence.

Cardano is an emerging 3rd generation blockchain and is continuously under development [10]. The Cardano community could benefit from this tool for rapid smart contract development for templates, prototypes, or production-ready contracts.

This chapter describes a new data model designed explicitly for Plutus contracts, underlying models for generating Plutus code, and parts of the Plutus generator itself.

3.1 Plutus Contract data model

For purposes of this generator, a new contract data model has been created. When generating a contract from the DasContract file, it first has to be converted into a PlutusContract format and then into a Plutus code.

This PlutusContract data model is an extra step with extra work; however, it brings a series of significant benefits:

1. The DasContract format is suitable for serialization, deserialization, databases, etc. Unfortunately, this compromises the data model with weaknesses, such as referring to an entity with an id instead of Object-Oriented (OO) native references (or pointers) [27]. This dramatically worsens working with DasContract objects. Suboptimal program interfaces occur, as broader, seemingly useless contexts, need to be passed into methods or classes. Algorithms are polluted with searching logic.

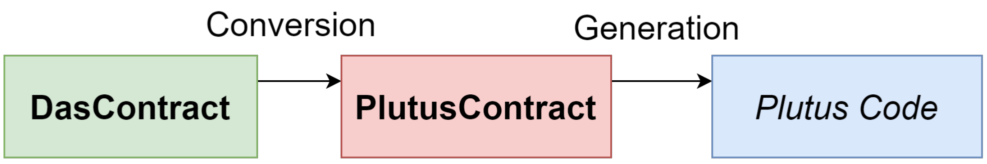


Figure 3.1: Hight abstraction of the conversion and generation process

PlutusContract does not need to be serialized and is properly modeled in an OO manner with language-specific features, such as object references.

2. DasContract carries a series of backward compatibility features that are not needed in the Plutus contract generator. PlutusContract is a cleaner data model.

3. DasContract provides irrelevant or strait up unwanted features (in the context of Plutus), such as the support for ERC-20 token models. PlutusContract will be a cleaner data model without these features.

4. DasContract does not provide needed features for the Plutus generator and does not implement suitable patterns for expanding its capabilities, such as the visitor pattern. Although other methods are available, such as C# extensions, they are not as ideal as the visitor pattern. PlutusContract can be set up in any suitable way.

5. PlutusContract will be designed recursively on viable places, such as the contracts' process, making the generation process simpler and more aligned with Haskell's purely functional approach.

6. If any changes to the DasContract occur, there is a probability that adjustments need to be made only at the data model conversion level. These changes are objectively much simpler than adjusting the Plutus contract generator and can be made by a person unfamiliar with the generators' inner workings.

Problems with the DasContract format in this use-case and improvements that PlutusContract implementation brings heavily outweigh the work required to implement the PlutusContract. High abstraction of the conversion and generation process is visualized at figure number 3.1.

The PlutusContract consists of three major sections, same as the DasContract:

- Users
- Data model
- Processes

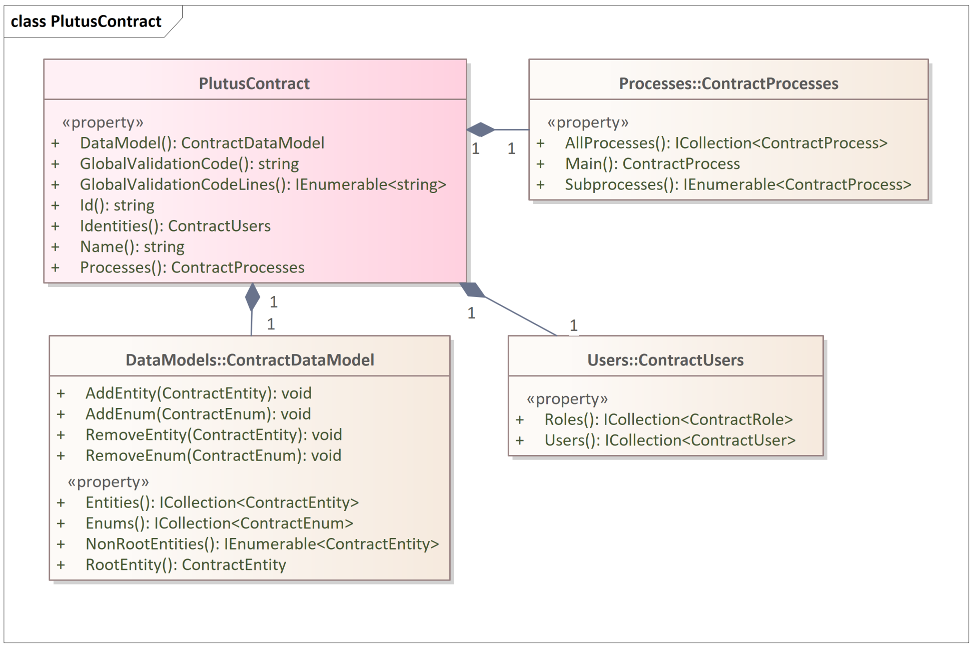

class PlutusContract
PlutusContract
«property»
+ DataModel(): ContractDataModel
+ GlobalValidationCode(): string
+ GlobalValidationCodeLines(): IEnumerable<string>
+ Id(): string
+ Identities(): ContractUsers
+ Name(): string
+ Processes(): ContractProcesses
Processes::ContractProcesses
«property»
+ AllProcesses(): ICollection<ContractProcess>
+ Main(): ContractProcess
+ Subprocesses(): IEnumerable<ContractProcess>
DataModels::ContractDataModel
+ AddEntity(ContractEntity): void
+ AddEnum(ContractEnum): void
+ RemoveEntity(ContractEntity): void
+ RemoveEnum(ContractEnum): void
«property»
+ Entities(): ICollection<ContractEntity>
+ Enums(): ICollection<ContractEnum>
+ NonRootEntities(): IEnumerable<ContractEntity>
+ RootEntity(): ContractEntity
Users::ContractUsers
«property»
+ Roles(): ICollection<ContractRole>
+ Users(): ICollection<ContractUser>
1
1
1
1
1
1

Figure 3.2: A UML class diagram of PlutusContract and its immediate members

It also contains a code for global contract validation, which is extracted from the single executable process. This code is run every state transition and provides more detailed tools, such as the EUTXO information. This is for advanced contracts with more complex logic that the DasContract format does not account for.

A UML class diagram of PlutusContract and its immediate members is pictured in figure number 3.2.

3.1.1 Users

The PlutusContract has modeled users very similarly to the DasContract. There are roles, which have a name and a description, and there are users who have a name, a description, a wallet public key, and a collection of their roles. A UML class diagram of users and roles can be seen in figure number 3.3.

3.1.2 Data model

The PlutusContract, same as DasContract, has a custom contract data model. The data model consists of Enums and Entities – Tokens are excluded from this data model since ERC-20 tokens, used in the DasContract, are irrelevant in the Cardano blockchain.

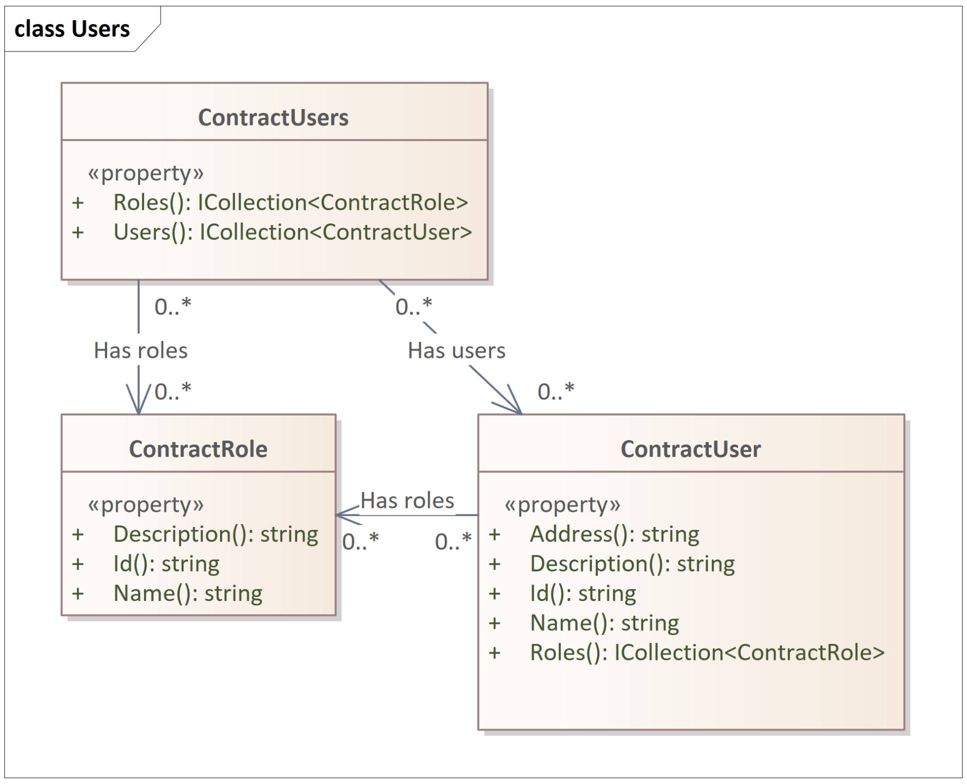


Figure 3.3: A UML class diagram of PlutusContract users

PlutusContract is similar in most areas except entity properties. PlutusContract chose a different approach when categorizing property types. Property types may be:

- A primitive property
- A reference property
- A dictionary property
- An enum property

DasContract crams these properties into one class, which results in awkward situations, where some property members are useless depending on other property members. It is counter-intuitive and requires a manual to know which members are useful and useless in what cases.

Plutus contract chose to distinct primitive, reference, enum, and dictionary properties resulting in proper OO design. No members are useless anymore. Some members are not even needed since the distinction clears up certain situations. Recursive structures are now possible in the case of dictionary

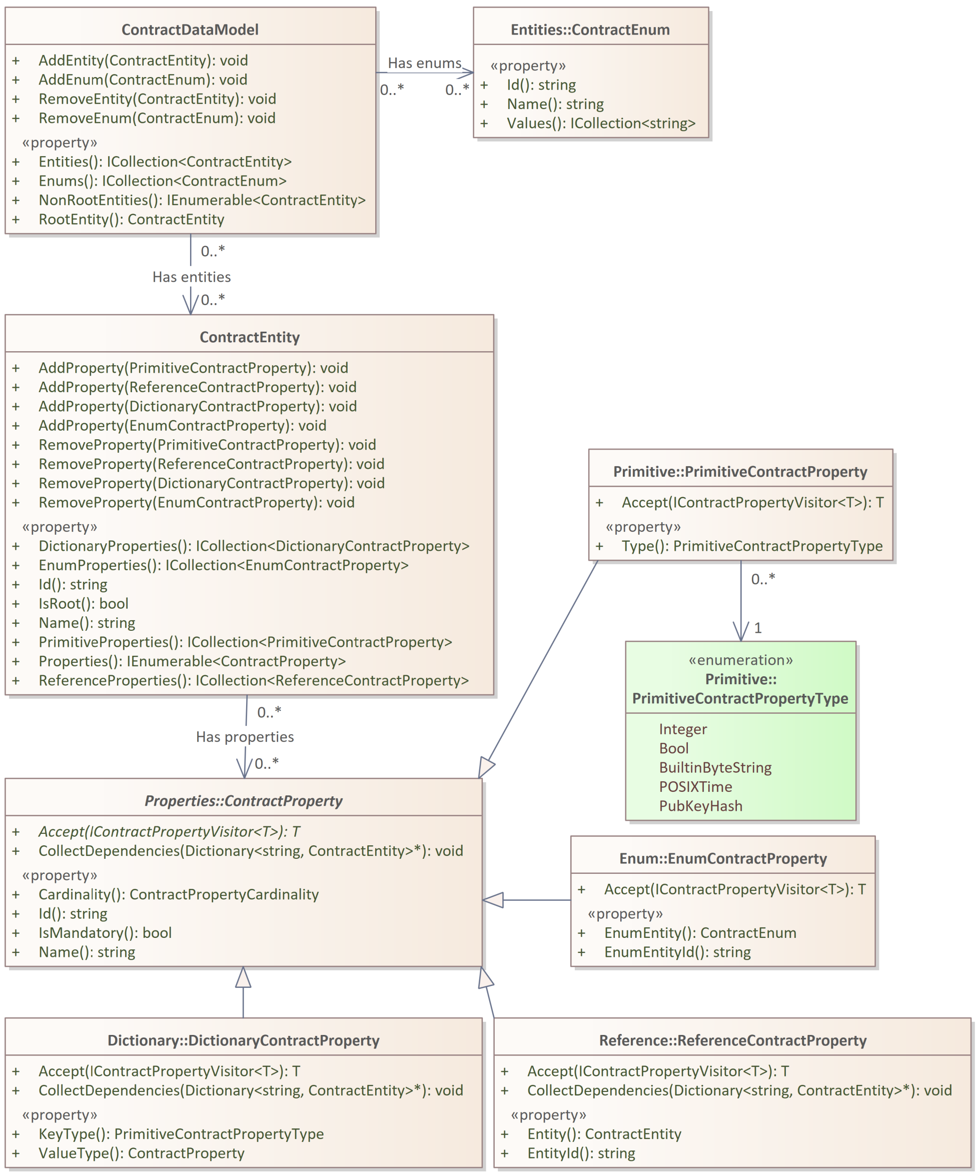

ContractDataModel
+ AddEntity(ContractEntity): void
+ AddEnum(ContractEnum): void
+ RemoveEntity(ContractEntity): void
+ RemoveEnum(ContractEnum): void
«property»
+ Entities(): ICollection<ContractEntity>
+ Enums(): ICollection<ContractEnum>
+ NonRootEntities(): IEnumerable<ContractEntity>
+ RootEntity(): ContractEntity
Has enums
0..*
0..*
Entities::ContractEnum
«property»
+ Id(): string
+ Name(): string
+ Values(): ICollection<string>
0..*
Has entities
0..*
ContractEntity
+ AddProperty(PrimitiveContractProperty): void
+ AddProperty(ReferenceContractProperty): void
+ AddProperty(DictionaryContractProperty): void
+ AddProperty(EnumContractProperty): void
+ RemoveProperty(PrimitiveContractProperty): void
+ RemoveProperty(ReferenceContractProperty): void
+ RemoveProperty(DictionaryContractProperty): void
+ RemoveProperty(EnumContractProperty): void
«property»
+ DictionaryProperties(): ICollection<DictionaryContractProperty>
+ EnumProperties(): ICollection<EnumContractProperty>
+ Id(): string
+ IsRoot(): bool
+ Name(): string
+ PrimitiveProperties(): ICollection<PrimitiveContractProperty>
+ Properties(): IEnumerable<ContractProperty>
+ ReferenceProperties(): ICollection<ReferenceContractProperty>
Primitive::PrimitiveContractProperty
+ Accept(IContractPropertyVisitor<T>): T
«property»
+ Type(): PrimitiveContractPropertyType
0..*
1
«enumeration»
Primitive::
PrimitiveContractPropertyType
Integer
Bool
BuiltinByteString
POSIXTime
PubKeyHash
0..*
Has properties
0..*
Properties::ContractProperty
+ Accept(IContractPropertyVisitor<T>): T
+ CollectDependencies(Dictionary<string, ContractEntity>*): void
«property»
+ Cardinality(): ContractPropertyCardinality
+ Id(): string
+ IsMandatory(): bool
+ Name(): string
Enum::EnumContractProperty
+ Accept(IContractPropertyVisitor<T>): T
«property»
+ EnumEntity(): ContractEnum
+ EnumEntityId(): string
Dictionary::DictionaryContractProperty
+ Accept(IContractPropertyVisitor<T>): T
+ CollectDependencies(Dictionary<string, ContractEntity>*): void
«property»
+ KeyType(): PrimitiveContractPropertyType
+ ValueType(): ContractProperty
Reference::ReferenceContractProperty
+ Accept(IContractPropertyVisitor<T>): T
+ CollectDependencies(Dictionary<string, ContractEntity>*): void
«property»
+ Entity(): ContractEntity
+ EntityId(): string

Figure 3.4: A UML class diagram of PlutusContract data models

property (for example, a dictionary of a dictionary of a dictionary). Enums, references, and dictionaries now refer to their related entities by reference, instead of ids, making them more accessible and more native to work with, as stated in the introduction of this section.

PlutusContract properties provide support for the visitor pattern in the case of need.

A UML class diagram of important data model classes can be seen in figure number 3.4.

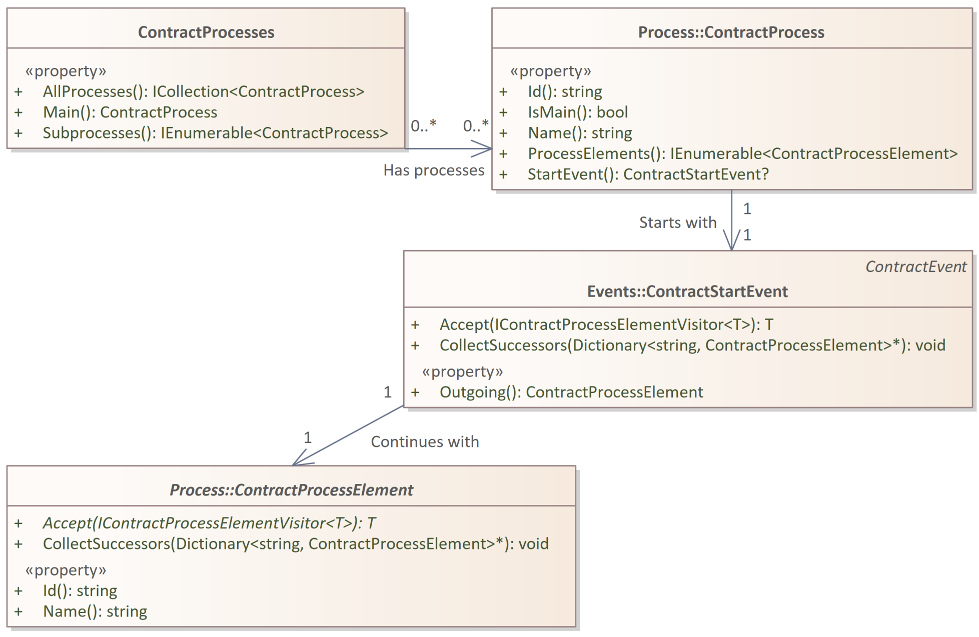

ContractProcesses
«property»
+ AllProcesses(): ICollection<ContractProcess>
+ Main(): ContractProcess
+ Subprocesses(): IEnumerable<ContractProcess>
0..*
0..*
Has processes
Process::ContractProcess
«property»
+ Id(): string
+ IsMain(): bool
+ Name(): string
+ ProcessElements(): IEnumerable<ContractProcessElement>
+ StartEvent(): ContractStartEvent?
1
Starts with
1
ContractEvent
Events::ContractStartEvent
+ Accept(IContractProcessElementVisitor<T>): T
+ CollectSuccessors(Dictionary<string, ContractProcessElement>*): void
«property»
+ Outgoing(): ContractProcessElement
1
1
Continues with
Process::ContractProcessElement
+ Accept(IContractProcessElementVisitor<T>): T
+ CollectSuccessors(Dictionary<string, ContractProcessElement>*): void
«property»
+ Id(): string
+ Name(): string

Figure 3.5: A UML class diagram of PlutusContract processes

3.1.3 Processes

PlutusContract processes have been mostly reworked. The core inheritance structure remains similar; however, the relationship structure is reinvigorated.

DasContract has processes with their respective lists of elements. These elements from the point of the BPMN have connections. DasContract handles these connections as extra `SequenceFlow` objects, where each `SequenceFlow` knows its source and destination. Boundary events are also elements that know only the id of an element they are attached to. This approach is more data-oriented than object-oriented and shows noticeable annoyances – polluted interfaces, algorithms, and other problems mentioned in the introduction of this section.

PlutusContract reworks the way elements are sequenced together with an undiluted OO approach. Every process contains the start event task. The start event then references the next element in the line, which references the next element in the line etc. The structure then becomes effortlessly recursively traversable. Working and generating on such structures is simpler and more native to OO programming. Task elements have their boundary events referenced directly. The only discovered negative side of this structure is difficult aggregating information from all elements – however, this has been resolved with a recursive aggregating algorithm and its implementation for potential future users. The process structure is described at a UML diagram number 3.5.

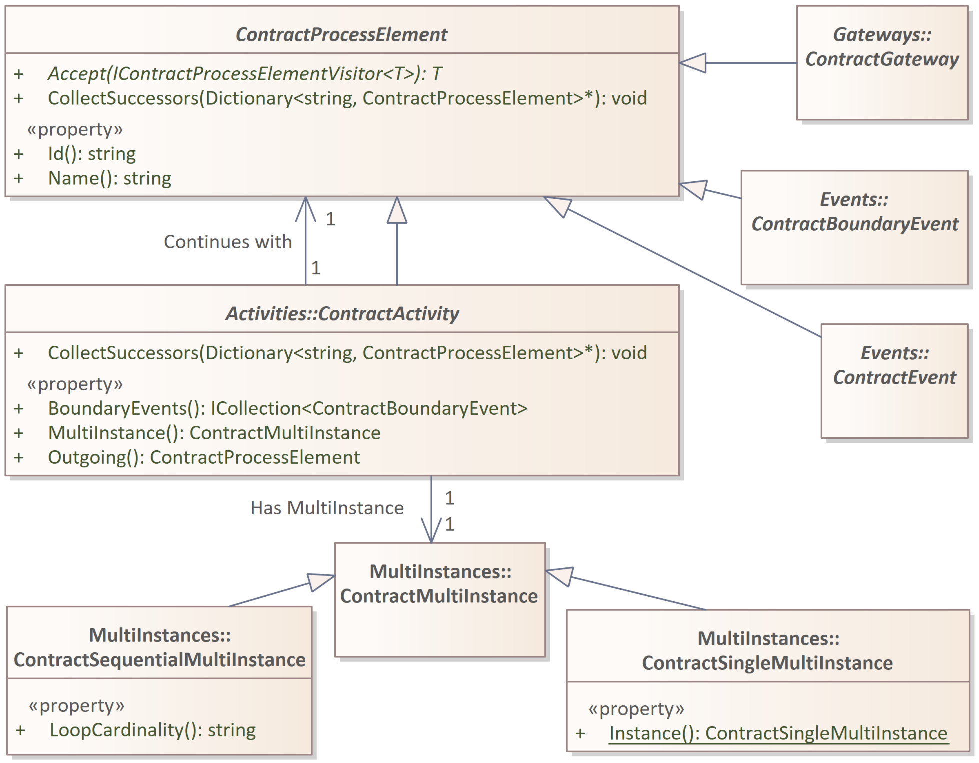

ContractProcessElement
+ Accept(IContractProcessElementVisitor<T>): T
+ CollectSuccessors(Dictionary<string, ContractProcessElement>*): void
«property»
+ Id(): string
+ Name(): string
Gateways::
ContractGateway
Events::
ContractBoundaryEvent
Events::
ContractEvent
Continues with
1
1
Activities::ContractActivity
+ CollectSuccessors(Dictionary<string, ContractProcessElement>*): void
«property»
+ BoundaryEvents(): ICollection<ContractBoundaryEvent>
+ MultiInstance(): ContractMultiInstance
+ Outgoing(): ContractProcessElement
Has MultiInstance
1
1
MultiInstances::
ContractMultiInstance
MultiInstances::
ContractSequentialMultiInstance
«property»
+ LoopCardinality(): string
MultiInstances::
ContractSingleMultiInstance
«property»
+ Instance(): ContractSingleMultiInstance

Figure 3.6: A UML class diagram of PlutusContract process elements

DasContract has similar problems, as the data model properties, with multi instances. DasContract tasks have members like `LoopCardinality`, `LoopCollection`, and `InstanceType`, which are sometimes needed and sometimes useless, depending on the `InstanceType` value. This bears the same hardships as in the case of multiple data model properties mushed inside one object, mentioned in section number 3.1.2.

PlutusContract has extra multi-instance objects removing the problem mentioned above in a proper OO manner, as modeled on a UML class diagram number 3.6.

3.1.4 PlutusContract summary

DasContract format is suitable for serialization and is supposed to be blockchain-independent; however, this makes proper OO modeling for Plutus contract generation rather difficult. Figure number 3.1 outlines a new bespoke data model PlutusContract that has been designed and created as an intermediate between the DasContract format and the generator.

The PlutusContract offers a series of significant advantages, such as:

- Some potential code updates become trivial

- The ideal structure for the code generator
- No unwanted features
- Added wanted blockchain-specific features

3.2 Plutus Code models

In the case of this thesis, smart contract code generation is not an easy endeavor. Proper data models, interfaces, and code structures for Plutus generation need to be created to build clean and consistent generator algorithms.

An interface that resonates throughout the project is the `INamable` interface, which contains only one property `Name`. In Haskell programming, many namable elements exist, such as functions, types, constructors, variables, etc. Usually, the name is the only dynamic thing needed to generate a code. For example, only the name is needed to invoke a parameterless method. Or, names of properties and names of their respective types are needed to generate a data structure.

First and foremost, the result of the generation is supposed to be a string-convertible code. The `IStringable` interface ensures using the `InString` method that any implementations will be able to convert into a string.

The generator perceives generated code as Plutus code, even though the code is essentially a Haskell code. Plutus code is perceived as a sequence of Plutus lines.

`IPlutusLine` is an interface that contains an indent value and inherits the `IStringable` interface. An abstract implementation of the `IPlutusLine`, `PlutusLine`, provides bases for any one-line Plutus constructs, such as a Plutus comment, Plutus signature, Plutus import, etc. The most used `PlutusLine` implementation is `PlutusRawLine`, where any text is considered as a line. It is the most used since many lines are unique or user-provided and can not be generalized.

`IPlutusCode` is an interface that contains `Append` and `Prepend` methods for aggregating other `IPlutusLine` and `IPlutusCode`. `IPlutusCode` is also `IStringable`. An implementation of `IPlutusCode`, `PlutusCode`, accepts an `IEnumerable` structure of `IPlutusLine`s and is capable of converting them into a multi-lined code string. `PlutusCode` is a parent of many useful structures, such as the Plutus function, various Plutus typeclass instances, Plutus data structures, etc. Another implementation of `IPlutusCode` is `PlutusCodes`. It accepts `IEnumerable` structure of `IPlutusCode` and is used for appending/prepending procedures.

The entirety of Plutus code models are **immutable** structures to promote propper OO approach and prevent any cascading errors encountered by modifying key properties, such as names.

Base Plutus models and interfaces can be seen in figure number 3.7.

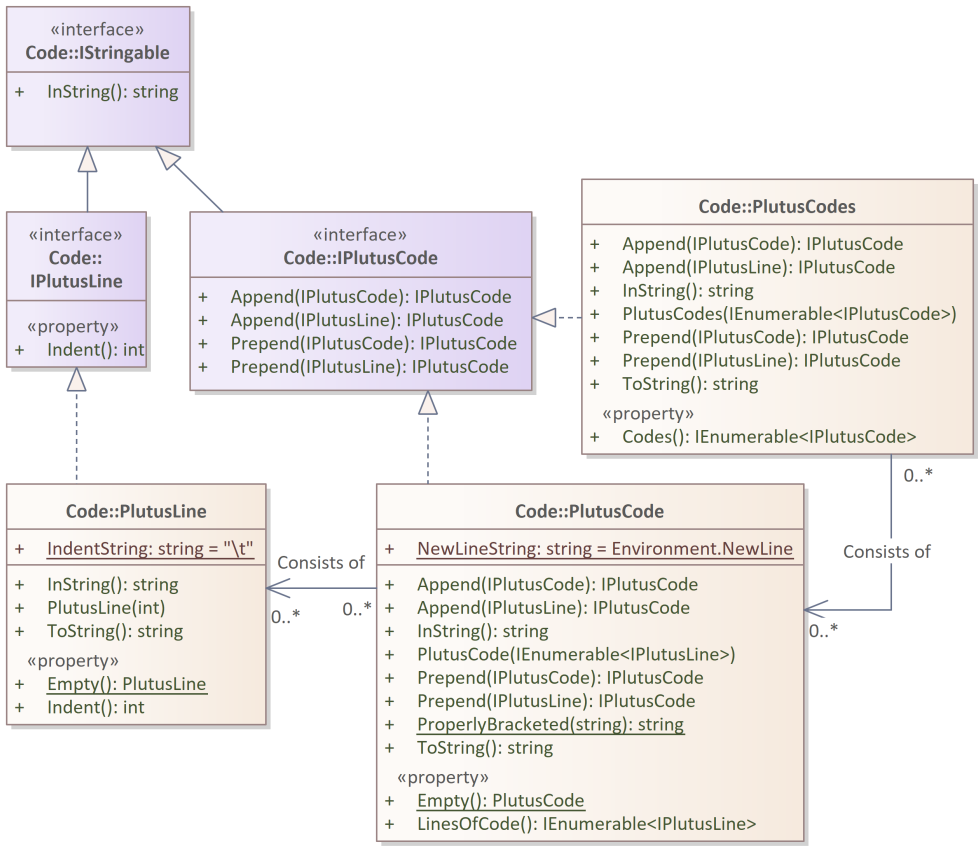

«interface»
Code::IStringable
+ InString(): string
«interface»
Code::IPlutusLine
«property»
+ Indent(): int
«interface»
Code::IPlutusCode
+ Append(IPlutusCode): IPlutusCode
+ Append(IPlutusLine): IPlutusCode
+ Prepend(IPlutusCode): IPlutusCode
+ Prepend(IPlutusLine): IPlutusCode
Code::PlutusCodes
+ Append(IPlutusCode): IPlutusCode
+ Append(IPlutusLine): IPlutusCode
+ InString(): string
+ PlutusCodes(IEnumerable<IPlutusCode>)
+ Prepend(IPlutusCode): IPlutusCode
+ Prepend(IPlutusLine): IPlutusCode
+ ToString(): string
«property»
+ Codes(): IEnumerable<IPlutusCode>
0..*
Consists of
0..*
Code::PlutusLine
+ IndentString: string = "\t"
+ InString(): string
+ PlutusLine(int)
+ ToString(): string
«property»
+ Empty(): PlutusLine
+ Indent(): int
Consists of
0..*
0..*
Code::PlutusCode
+ NewLineString: string = Environment.NewLine
+ Append(IPlutusCode): IPlutusCode
+ Append(IPlutusLine): IPlutusCode
+ InString(): string
+ PlutusCode(IEnumerable<IPlutusLine>)
+ Prepend(IPlutusCode): IPlutusCode
+ Prepend(IPlutusLine): IPlutusCode
+ ProperlyBracketed(string): string
+ ToString(): string
«property»
+ Empty(): PlutusCode
+ LinesOfCode(): IEnumerable<IPlutusLine>

Figure 3.7: A UML class diagram of Plutus code data models

3.2.1 Types

Since Haskell (Plutus) is type-safe, types need to be considered when modeling and generating the resulting code [14]. The Plutus code data model looks at types from three perspectives:

- Predetermined
- Predefined
- User-defined

Predetermined types are essential types that will exist in the resulting Plutus code. For example, the `Datum` data type will indeed always eventually exist; thus, it is predetermined. This serves as a binding glue for referencing across multiple components. Especially datum is needed almost everywhere, so instead of passing tons of arguments for type names or type objects, these

Listing 3.1 Example of a Plutus type model

```
public abstract class PlutusPremadeType : INamable
{
    public abstract string Name { get; }
}

public class PlutusContractDatum : PlutusPremadeType
{
    public override string Name { get; } = "ContractDatum";

    public static PlutusContractDatum Type { get; }
        = new PlutusContractDatum();
}
```

Listing 3.2 Plutus list type model

```
public class PlutusList : PlutusPremadeType
{
    public PlutusList(INamable innerType)
    {
        InnerType = innerType;
    }

    public INamable InnerType { get; }

    public override string Name => $"[{InnerType.Name}]";

    public static PlutusList Type(INamable innerType)
        => new PlutusList(innerType);
}
```

predetermined objects have been created and are available anywhere. A predetermined object is essentially a name carried in an OO approach, as seen on listing number 3.1.

Predefined types are types that already exist, such as an Integer, a Bool, a String, etc. They are modeled the exact same way as predetermined types with an example on listing number 3.1. Some predefined types require other types; for example, a list needs to be a list of something, as seen on listing number 3.2.

User-defined types are types defined with the `data` or `newtype` keyword. These keywords are very versatile, and covering all possible situations would be too much effort. Two essential situations were identified and implemented:

Listing 3.3 Haskell example of an algebraic data type and a record data type

```haskell
-- Algebraic data type
data ContractState =
        MainProcessStart |
        ...
        SuccessWithdrawalProcess SuccessWithdrawalProcess |
        ContractFinished
  deriving (Show, Generic, FromJSON, ToJSON)

-- Record data type
data Role = Role {
        rName :: BuiltinByteString,
        rDescription :: BuiltinByteString
} deriving (Show, Generic, FromJSON, ToJSON)
```

algebraic types and **records**. An algebraic type is a data type with several constructors and their parameters. A record is a data type with one single constructor and multiple parameters; however, functions that extract these parameters are automatically generated using a special Haskell record syntax [35]. An example of these types is at listing number 3.3. A UML class diagram of their Plutus solutions can be found at figure 3.8.

Apart from these three categories, there are also types for miscellaneous situations where the type is not easy to model and is simply written in string. However, these situations are avoided as much as possible.

3.2.2 Functions

A purely functional language can not be developed without functions. Three function use-cases have been modeled and used in the generation process:

- Function
- One-line function
- Guard function

Additionally, every function has a signature. The signature consists of a name and parameter types.

A function is a multiline Plutus code, requiring a signature, parameter names, and implementation Plutus lines.

A one-line function is a shortened version of the function, where the entire function definition is on a single line.

Guard function uses the Haskell guard syntax.

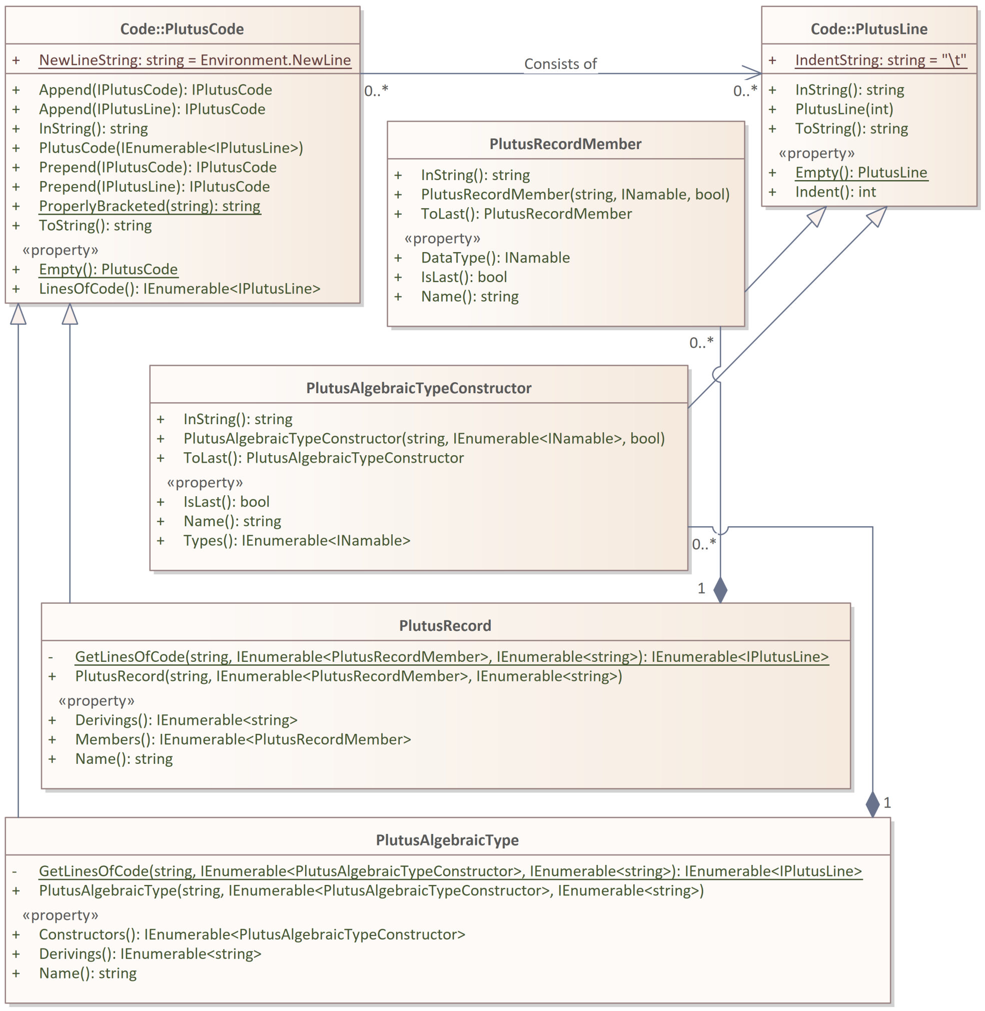

Code::PlutusCode
+ NewLineString: string = Environment.NewLine
+ Append(IPlutusCode): IPlutusCode
+ Append(IPlutusLine): IPlutusCode
+ InString(): string
+ PlutusCode(IEnumerable<IPlutusLine>)
+ Prepend(IPlutusCode): IPlutusCode
+ Prepend(IPlutusLine): IPlutusCode
+ ProperlyBracketed(string): string
+ ToString(): string
«property»
+ Empty(): PlutusCode
+ LinesOfCode(): IEnumerable<IPlutusLine>
Consists of
0..*
0..*
Code::PlutusLine
+ IndentString: string = "\t"
+ InString(): string
+ PlutusLine(int)
+ ToString(): string
«property»
+ Empty(): PlutusLine
+ Indent(): int
PlutusRecordMember
+ InString(): string
+ PlutusRecordMember(string, INamable, bool)
+ ToLast(): PlutusRecordMember
«property»
+ DataType(): INamable
+ IsLast(): bool
+ Name(): string
0..*
PlutusAlgebraicTypeConstructor
+ InString(): string
+ PlutusAlgebraicTypeConstructor(string, IEnumerable<INamable>, bool)
+ ToLast(): PlutusAlgebraicTypeConstructor
«property»
+ IsLast(): bool
+ Name(): string
+ Types(): IEnumerable<INamable>
0..*
1
PlutusRecord
- GetLinesOfCode(string, IEnumerable<PlutusRecordMember>, IEnumerable<string>): IEnumerable<IPlutusLine>
+ PlutusRecord(string, IEnumerable<PlutusRecordMember>, IEnumerable<string>)
«property»
+ Derivings(): IEnumerable<string>
+ Members(): IEnumerable<PlutusRecordMember>
+ Name(): string
1
PlutusAlgebraicType
- GetLinesOfCode(string, IEnumerable<PlutusAlgebraicTypeConstructor>, IEnumerable<string>): IEnumerable<IPlutusLine>
+ PlutusAlgebraicType(string, IEnumerable<PlutusAlgebraicTypeConstructor>, IEnumerable<string>)
«property»
+ Constructors(): IEnumerable<PlutusAlgebraicTypeConstructor>
+ Derivings(): IEnumerable<string>
+ Name(): string

Figure 3.8: A UML class diagram of Plutus user-defined types

3.2.3 Typeclass instances

The generated code and its user-defined data types need to implement certain type classes. For this purpose, several instance generators were thought out and implemented.

The Plutus models currently support four instances. `PlutusMakeLift` and `PlutusUnstableMakeIsData` are just one-line referrers to instance generation functions inside a Plutus library. `PlutusEq` and `PlutusDefault` generate equality and default value instances. These are directly derived from user-defined data types.

The `PlutusEq` generates `Eq` instance for a Plutus record or Plutus algebraic type. It uses a simple pattern to match each constructor and its values, as pictured on listing number 3.4. The `Eq` must be marked with `INLINABLE`

Listing 3.4 Haskell example of a records' `Eq` instance

```
instance Eq Role where
    {-# INLINABLE (==) #-}
    Role a b == Role a' b' = (a == a') && (b == b')
```

Listing 3.5 Haskell example of a records' `Default` instance

```
instance Default Role where
    {-# INLINABLE def #-}
    def = Role {
        rName = "",
        rDescription = ""
    }
```

pragma to property compile into the Plutus Core (=blockchain code).

The `PlutusDefault` generates `Default` instance for a Plutus record or Plutus algebraic type. It figures out the default value of a type and then assigns it. If it needs to generate default value for an algebraic type, the default constructor needs to be supplied. A default instance is pictured on listing number 3.5. The `Default` must be marked with `INLINABLE` pragma to property compile into the Plutus Core (=blockchain code).

Instance generators highlight the advantages of the proper OO approach to Plutus code generation instead of imperatively appending strings together – making these tools challenging to implement. Instances are an enormous time saver and are very convenient.

3.3 Plutus contract generator

With PlutusContract data model – a data model suitable for Plutus contract generation – and support models for Plutus code generation, a quality Plutus contract generator could be built.

The base interface for any generator is `ICodeGenerator` with a single method `Generate`. This method returns `IPlutusCode`.

The resulting product, capable of generating the entire Plutus contract, is the `PlutusContractGenerator`. This generator consists of six other generators, each ensuring generation of different parts of the contract. These parts are:

- `PlutusContractPragmaGenerator` generates pragma expressions for the contract.
- `PlutusContractModuleGenerator` generates the module definition for the contract.

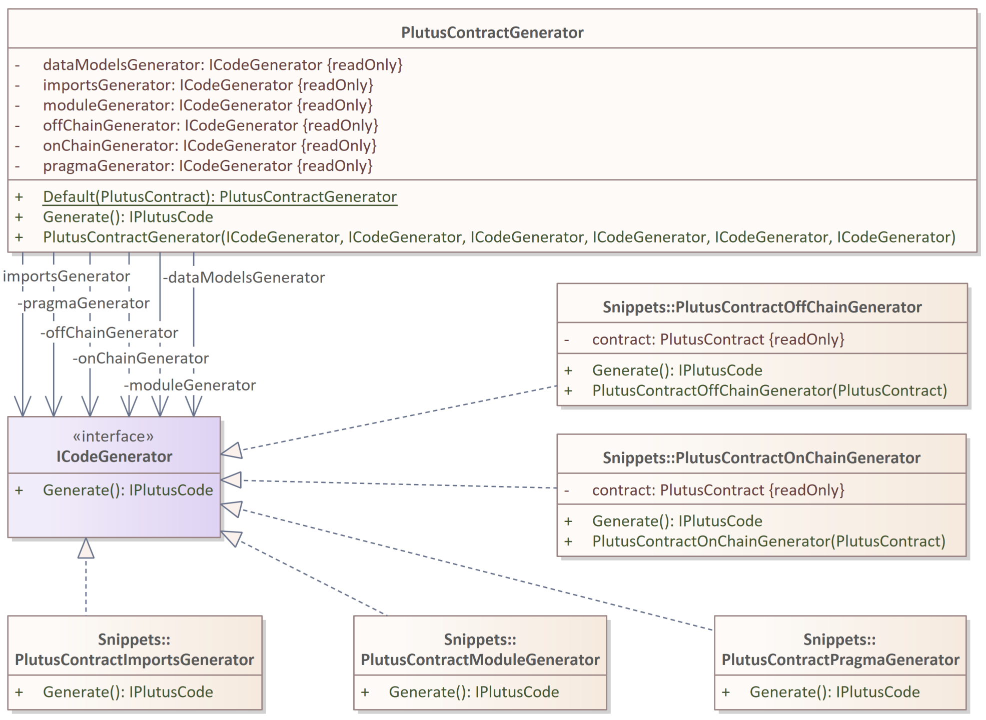

PlutusContractGenerator
- dataModelsGenerator: ICodeGenerator {readOnly}
- importsGenerator: ICodeGenerator {readOnly}
- moduleGenerator: ICodeGenerator {readOnly}
- offChainGenerator: ICodeGenerator {readOnly}
- onChainGenerator: ICodeGenerator {readOnly}
- pragmaGenerator: ICodeGenerator {readOnly}
+ Default(PlutusContract): PlutusContractGenerator
+ Generate(): IPlutusCode
+ PlutusContractGenerator(ICodeGenerator, ICodeGenerator, ICodeGenerator, ICodeGenerator, ICodeGenerator, ICodeGenerator)
-importsGenerator
-dataModelsGenerator
-pragmaGenerator
-offChainGenerator
-onChainGenerator
-moduleGenerator
«interface»
ICodeGenerator
+ Generate(): IPlutusCode
Snippets::PlutusContractOffChainGenerator
- contract: PlutusContract {readOnly}
+ Generate(): IPlutusCode
+ PlutusContractOffChainGenerator(PlutusContract)
Snippets::PlutusContractOnChainGenerator
- contract: PlutusContract {readOnly}
+ Generate(): IPlutusCode
+ PlutusContractOnChainGenerator(PlutusContract)
Snippets::
PlutusContractImportsGenerator
+ Generate(): IPlutusCode
Snippets::
PlutusContractModuleGenerator
+ Generate(): IPlutusCode
Snippets::
PlutusContractPragmaGenerator
+ Generate(): IPlutusCode

Figure 3.9: A UML class diagram of the Plutus generator

- `PlutusContractImportsGenerator` generates import expressions for the contract.

- `PlutusContractDataModelGenerator` generates data models for the contract, including phases, datum, forms, redeemers, users, and contract parameter.

- `PlutusContractOnChainGenerator` generates code which is then compiled and run on the blockchain or is related to on-chain code.

- `PlutusContractOffChainGenerator` generates mostly endpoints for the contract, which run inside users wallet.

The code is generated in a user-readable manner. Indentation and white spaces are built to be similar to *"traditional"* hand-made programs. Sections, subsections, transitions, and even some code lines are well commented. The resulting contract should be easily modifiable, as real hand-made contracts were made as templates for the contract generator.

Relationships of generators are visualized as UML class diagram in figure number 3.9.

Listing 3.6 A snippet of the pragma generator code

```
public IPlutusCode Generate()
{
    var pragmas = new PlutusCode(new List<IPlutusLine>()
    {
        new PlutusPragma(0, "LANGUAGE DataKinds"),
        ...
        new PlutusPragma(0, "LANGUAGE TypeOperators"),
        PlutusLine.Empty,
        new PlutusPragma(0,
            "OPTIONS_GHC -fno-warn-unused-imports"),
        PlutusLine.Empty,
    });

    return pragmas;
}
```

3.3.1 Pragma generator

The pragma generator is static and trivial. It generates several helpful or required pragma statement lines. A snippet of the pragma generator can be seen at listing number 3.6.

3.3.2 Module generator

The Haskell module, where the contract is generated, has been named PlutusContract. Haskell modules allow setting what elements of the modules are exported (public). In the case of PlutusContract, everything has been set public.

A proposition for setting public only certain elements is described in section 5.3.5 with a description of why everything is currently set public.

A snippet of the module generator can be seen at listing number 3.7.

3.3.3 Imports generator

The imports generator is static and trivial. It generates required import statement lines. A snippet of the import generator can be seen at listing number 3.8.

3.3.4 Data model generator

The data model generator is responsible for multiple subsections of the contract, primarily related to defining data types and operations with these data types.

Listing 3.7 A snippet of the module generator code

```
public IPlutusCode Generate()
{
    var module = new PlutusCode(new List<IPlutusLine>()
    {
        new PlutusRawLine(0, "module PlutusContract"),
            new PlutusRawLine(1, "(module PlutusContract)"),
            new PlutusRawLine(1, "where"),

        PlutusLine.Empty,
        PlutusLine.Empty,
    });

    return module;
}
```

Listing 3.8 A snippet of the import generator code

```
public IPlutusCode Generate()
{
    var imports = new PlutusCode(new List<IPlutusLine>()
    {
        new PlutusImport(0, "Control.Monad hiding (fmap)"),
        new PlutusImport(0, "Data.Aeson (ToJSON, FromJSON)"),
        ...
        new PlutusImport(0, "Plutus.V1.Ledger.Bytes (fromHex)"),
        new PlutusImport(0, "Data.String"),
        PlutusLine.Empty,
    });

    return imports;
}
```

`SequentialMultiInstance` is a generated algebraic data type with two constructors: `ToLoop Int` and `LoopEnded`. This data type is assigned to contract phases with sequential multi-instance attached to them. It keeps the state of the loop – if it is still running or if it already ended. The equality instance is also generated. Additional functions are generated for data operations:

- `nextLoop` returns decreased multi-instance by one or sets it to the end.
- `toSeqMultiInstance` returns multi-instance based on the input number.
- `toNextSeqMultiInstance` combines the previous two functions.

Phases for the contract are generated. Firstly, all subprocesses are generated. Each subprocess is represented by an algebraic data type where each constructor is a process elements' name. If an element is also sequential-multi instance, the constructor then contains the `SequentialMultiInstance` value.

After all subprocesses have been generated, a data type for the main process is generated and named `ContractState`. The `ContractState` data type additionally contains `ContractFinished` state to finish the contract and a constructor for each subprocess and its states.

Equality instances are also generated for each subprocess and the main process. The default instance is generated for the main process, where the default constructor is set to the start event.

The **datum** for the contract is generated. Datum refers to the persistent data model described in PlutusContract. Firstly, enums are generated. Secondly, all entities in the data model are generated, except the root entity. Entities are generated in the order defined by topological sort, where reference property dependencies represent connections between nodes. This ensures that a type referring to an entity is not defined earlier than the entity itself, and by the nature of topological sort, it forbids any circular dependencies between entities. At last, the datum (=root entity) is generated with extra properties for keeping the current contracts' state and call stack.

Equality instance and default instance are also generated for each enum and entity.

Additional methods for the datum are generated:

- `pushState` that returns the datum with a new state in the call stack
- `popState` that returns a tuple of datum without the popped state in the call stack and the popped state

User form models are generated, where each user form record has corresponding properties to its form. Equality instances are also generated for each form.

Redeemers are generated. There is a redeemer constructor for each user activity with its respective form. Additional redeemers are always generated, specifically the `TimeoutRedeemer` for confirming user task timeouts and the `ContractFinishedRedeemer` for requesting the end of the contract. Equality instance is also generated for the redeemer.

User roles and users are generated with respective search methods for easy access. Equality instances and default instances are also generated for user roles and users. Additionally, a list of users and roles is generated.

Contract parameter is generated. Currently, the contract parameter contains a list of users, a list of roles, a default user, a default role, and a thread token.

3.3.5 On-chain generator

On-chain generator firstly generates **user form validations**. Implementations for the user form validations are extracted from the user activity validation codes. Form validation is generated as a pattern matching, where the redeemer is matched. The default validation is always False.

On-chain generator then generates two major sections: non-transactional transitions and transactional transitions.

The generator classifies process elements into three categories:

- Transactional (Tx)
- Non-Transactional (NonTx)
- Implicit

Transactional (Tx) states require a blockchain transaction to enter the state. The current list of Tx states is:

- end event in the root process – ends the entire contract and requires a transaction to release any unused funds if necessary
- user activity that is not a sequential-multi instance – user activities require submission of forms and/or funds which inherently needs a transaction
- timer boundary events – requires a transaction in order to ensure a time constraint

Non-Transactional (NonTx) states do not require a blockchain transaction to enter the state. They are executed before and after every transaction – every time a transaction is submitted, all possible Non-Transactional (NonTx) transitions are applied seamlessly. The current list of NonTx states is:

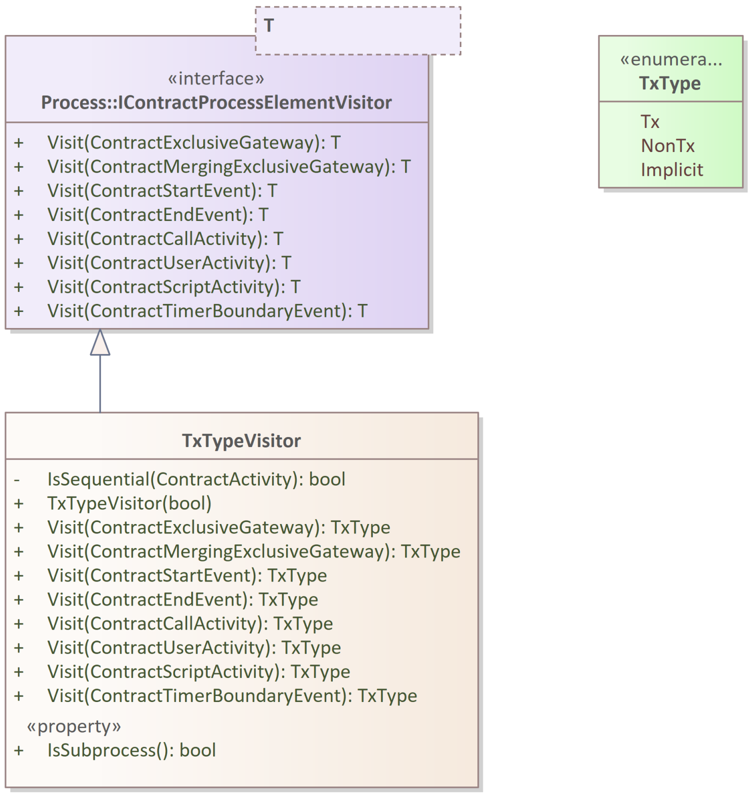

T
«interface»
Process::IContractProcessElementVisitor
+ Visit(ContractExclusiveGateway): T
+ Visit(ContractMergingExclusiveGateway): T
+ Visit(ContractStartEvent): T
+ Visit(ContractEndEvent): T
+ Visit(ContractCallActivity): T
+ Visit(ContractUserActivity): T
+ Visit(ContractScriptActivity): T
+ Visit(ContractTimerBoundaryEvent): T
«enumera...
TxType
Tx
NonTx
Implicit
TxTypeVisitor
- IsSequential(ContractActivity): bool
+ TxTypeVisitor(bool)
+ Visit(ContractExclusiveGateway): TxType
+ Visit(ContractMergingExclusiveGateway): TxType
+ Visit(ContractStartEvent): TxType
+ Visit(ContractEndEvent): TxType
+ Visit(ContractCallActivity): TxType
+ Visit(ContractUserActivity): TxType
+ Visit(ContractScriptActivity): TxType
+ Visit(ContractTimerBoundaryEvent): TxType
«property»
+ IsSubprocess(): bool

Figure 3.10: A UML class diagram of the Tx type visitor

- Branching exclusive gateway
- Merging exclusive gateway
- End event which is not in the main process
- Call activity
- User activity which is a sequential multi-instance
- Script activity

Implicit states currently include only the start event. These states are already set without any need to interfere.

This categorization classifies states based on the entry to the state, not what happens after or before. For example, all sequential multi-instance elements are always NonTx, because the entry to the state is NonTx. It does not tell anything about what happens in the loop or whether the iterations are transactional or non-transactional.

The categorization dramatically simplifies the generation of transitions since normally there would be n possible output connections from every element – which is n^2 possible situations to handle. This limits the number of potential cases to $3n$. In reality, it is only $2n$ since implicits are, by their nature, ignored. Additionally, most of the connections behave precisely the same, decreasing the number of different cases even further.

The on-chain generator uses two visitors to recursively traverse the entire process and collect the resulting code for transitions. Every visited element generates and returns a code appended with a visit of the next element.

One transition visitor is strictly a NonTx visitor. NonTx visits are always a simple datum transformation; however, there are more NonTx elements. The other transition visitor is strictly a Tx visitor. Tx visits are more complicated since they provide more options, such as value transfer, timeouts, etc.

After transitions, an **extra validator** is generated. The validator handles additional custom validation using the `GlobalValidation` code line.

An **end function** that determines the end of the contract is also generated. Contract ends when the state reaches `ContractFinished`.

A **state machine** that drives this contract is generated. The generated Plutus smart contract uses a state machine library to model states and transitions since the process flow is very similar to that of a state machine.

Additional boilerplate functions are generated.

3.3.6 Off-chain generator

Off-chain generator firstly generates a series of useful methods:

- `mapErr` is a wrapper and helps to format errors
- `createContractParam` accepts a thread token and creates a `Contract` monad with a new `ContractParam`
- `initContractParam` creates a thread token and continues to call the `createContractParam` function
- `onChainDatum` retrieves the current on-chain datum as a `Contract` monad using a state machine client
- `onChainValue` retrieves the current on-chain value as a `Contract` monad using a state machine client
- `logOnChainDatum` retrieves the on-chain datum and logs it
- `validateInputForm` validates an input form in a redeemer

These monad functions serve the next section of the off-chain generation – endpoints. Endpoints are functions that perform a task on the contract.

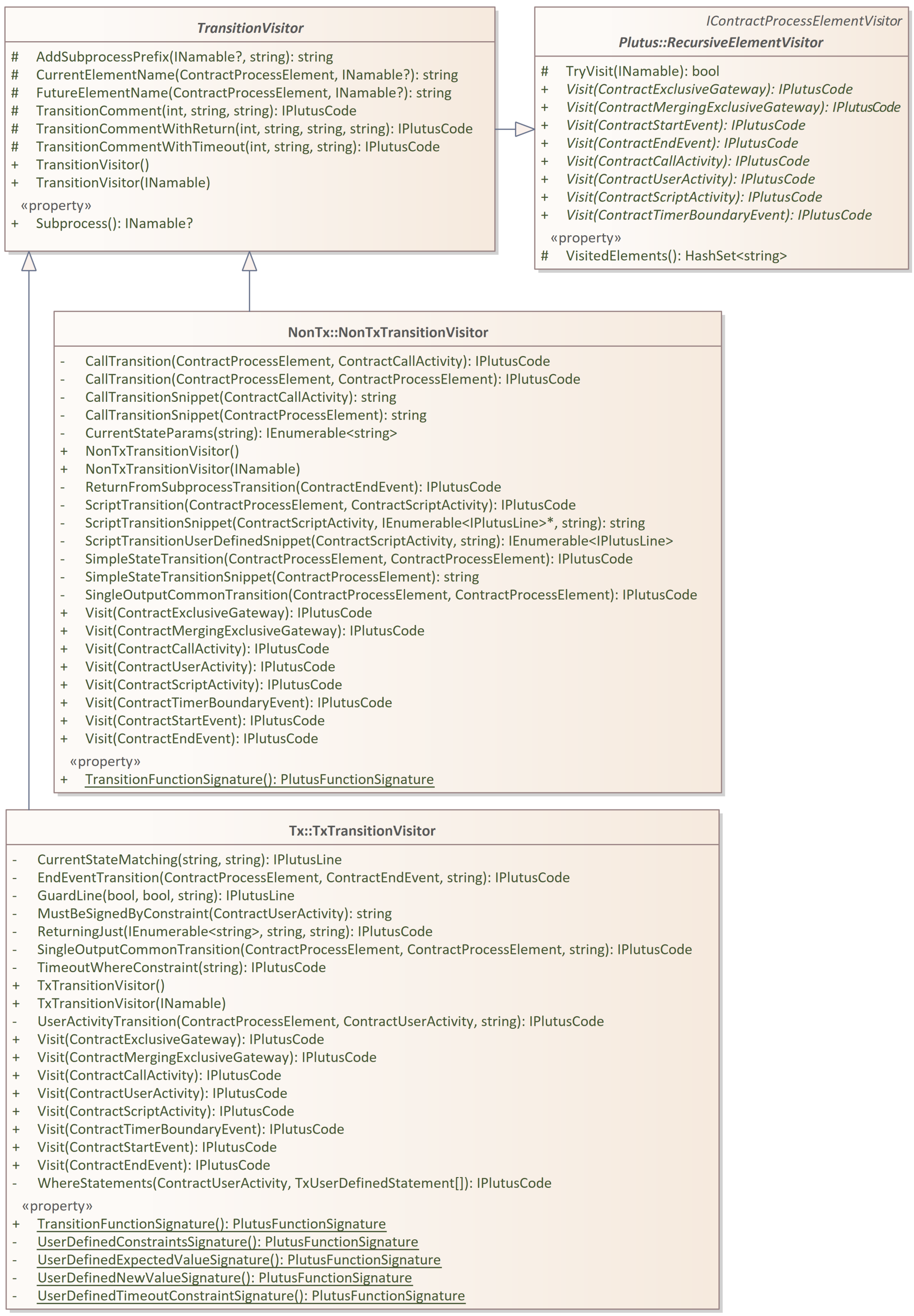

TransitionVisitor
# AddSubprocessPrefix(INamable?, string): string
# CurrentElementName(ContractProcessElement, INamable?): string
# FutureElementName(ContractProcessElement, INamable?): string
# TransitionComment(int, string, string): IPlutusCode
# TransitionCommentWithReturn(int, string, string, string): IPlutusCode
# TransitionCommentWithTimeout(int, string, string): IPlutusCode
+ TransitionVisitor()
+ TransitionVisitor(INamable)
«property»
+ Subprocess(): INamable?
IContractProcessElementVisitor
Plutus::RecursiveElementVisitor
# TryVisit(INamable): bool
+ Visit(ContractExclusiveGateway): IPlutusCode
+ Visit(ContractMergingExclusiveGateway): IPlutusCode
+ Visit(ContractStartEvent): IPlutusCode
+ Visit(ContractEndEvent): IPlutusCode
+ Visit(ContractCallActivity): IPlutusCode
+ Visit(ContractUserActivity): IPlutusCode
+ Visit(ContractScriptActivity): IPlutusCode
+ Visit(ContractTimerBoundaryEvent): IPlutusCode
«property»
# VisitedElements(): HashSet<string>
NonTx::NonTxTransitionVisitor
- CallTransition(ContractProcessElement, ContractCallActivity): IPlutusCode
- CallTransition(ContractProcessElement, ContractProcessElement): IPlutusCode
- CallTransitionSnippet(ContractCallActivity): string
- CallTransitionSnippet(ContractProcessElement): string
- CurrentStateParams(string): IEnumerable<string>
+ NonTxTransitionVisitor()
+ NonTxTransitionVisitor(INamable)
- ReturnFromSubprocessTransition(ContractEndEvent): IPlutusCode
- ScriptTransition(ContractProcessElement, ContractScriptActivity): IPlutusCode
- ScriptTransitionSnippet(ContractScriptActivity, IEnumerable<IPlutusLine>*, string): string
- ScriptTransitionUserDefinedSnippet(ContractScriptActivity, string): IEnumerable<IPlutusLine>
- SimpleStateTransition(ContractProcessElement, ContractProcessElement): IPlutusCode
- SimpleStateTransitionSnippet(ContractProcessElement): string
- SingleOutputCommonTransition(ContractProcessElement, ContractProcessElement): IPlutusCode
+ Visit(ContractExclusiveGateway): IPlutusCode
+ Visit(ContractMergingExclusiveGateway): IPlutusCode
+ Visit(ContractCallActivity): IPlutusCode
+ Visit(ContractUserActivity): IPlutusCode
+ Visit(ContractScriptActivity): IPlutusCode
+ Visit(ContractTimerBoundaryEvent): IPlutusCode
+ Visit(ContractStartEvent): IPlutusCode
+ Visit(ContractEndEvent): IPlutusCode
«property»
+ TransitionFunctionSignature(): PlutusFunctionSignature
Tx::TxTransitionVisitor
- CurrentStateMatching(string, string): IPlutusLine
- EndEventTransition(ContractProcessElement, ContractEndEvent, string): IPlutusCode
- GuardLine(bool, bool, string): IPlutusLine
- MustBeSignedByConstraint(ContractUserActivity): string
- ReturningJust(IEnumerable<string>, string, string): IPlutusCode
- SingleOutputCommonTransition(ContractProcessElement, ContractProcessElement, string): IPlutusCode
- TimeoutWhereConstraint(string): IPlutusCode
+ TxTransitionVisitor()
+ TxTransitionVisitor(INamable)
- UserActivityTransition(ContractProcessElement, ContractUserActivity, string): IPlutusCode
+ Visit(ContractExclusiveGateway): IPlutusCode
+ Visit(ContractMergingExclusiveGateway): IPlutusCode
+ Visit(ContractCallActivity): IPlutusCode
+ Visit(ContractUserActivity): IPlutusCode
+ Visit(ContractScriptActivity): IPlutusCode
+ Visit(ContractTimerBoundaryEvent): IPlutusCode
+ Visit(ContractStartEvent): IPlutusCode
+ Visit(ContractEndEvent): IPlutusCode
- WhereStatements(ContractUserActivity, TxUserDefinedStatement[]): IPlutusCode
«property»
+ TransitionFunctionSignature(): PlutusFunctionSignature
- UserDefinedConstraintsSignature(): PlutusFunctionSignature
- UserDefinedExpectedValueSignature(): PlutusFunctionSignature
- UserDefinedNewValueSignature(): PlutusFunctionSignature
- UserDefinedTimeoutConstraintSignature(): PlutusFunctionSignature

Figure 3.11: A UML class diagram of the transition generators

Listing 3.9 A snippet of an endpoint

```
withdrawTaskEndpoint :: (WithdrawTaskForm, ThreadToken)
    -> Contract w s Text ()
withdrawTaskEndpoint (form, threadToken) = do

        logInfo @String "withdrawTaskEndpoint called"

        -- Client setup
        contractParam <- createContractParam threadToken
        let client = contractClient contractParam
        logInfo @String "--- client created"

        -- Create redeemer
        let redeemer = WithdrawTaskRedeemer form

        -- Validate form
        validateInputForm threadToken client redeemer

        -- State transition
        void $ mapErr $ runStep client redeemer
        logOnChainDatum client
        logInfo @String "--- transition finished"

        logInfo @String "withdrawTaskEndpoint ended"
```

An endpoint first initializes a client state machine, which then can create and submit a new transaction to the blockchain.

Firstly, the contract must be initialized using the initialization endpoint. This will create the starting transaction in its implicit state. The initialization endpoint shares a thread token, which identifies the process and is essentially an NFT.

Currently, there is one endpoint for each user task. User task endpoints accept the identification thread token and a submission form. Then, the state machine client is set up and fed the appropriate redeemer, which results in a new submitted transaction if circumstances are correct (the form is valid, the state of the contract is proper, etc.). An example of an endpoint can be seen at listing number 3.9.

Additionally, endpoints for clearing timeouts and finishing the contract are created.

After all endpoints are created, the contracts' schema is generated. The contract schema is just a combined type of all endpoints.

The final matter of contract generation is the endpoints variable, which returns a `Contract` monad containing all endpoints (also `Contract` monads) ”together”. This variable represents the entire contract with all of its endpoints, and all endpoint calls are executed on this monad.

3.4 Testing

Testing is a vital part of any development. The proof-of-concept generator has been thoroughly tested; however, many of these tests are only systematic manual tests. The DasContract format is still evolving, and the expected time to develop adaptable and practical unit tests would be enormous. A *”playground”* contract has been designed to contain most of the implemented features in the generator to serve as a quicker manual-testing tool.

The DasContract-to-PlutusContract converter has been tested manually on exhaustive series of contracts and situations, including the playground contract. The convertor is rudimentary, and unit tests could be easily added in the future. However, this endeavor would be considerably time-consuming.

As a codebase for code generating, the Plutus code models have respective unit tests implemented. These models are vital for the more complex generator components, and any bugs occurring during the development process would cause further damage and time-consuming bug searches. Creating automated unit tests for these models has been a time-saver.

The proof-of-concept generator components have been tested manually and systematically. Unit testing of these components would require considerable effort and time, as described in section number 5.3.2.1.

3.5 NuGets

The Plutus smart contract generator is separated into two projects:

- **DasContract.Blockchain.Plutus.Data** contains the PlutusContract data model and the DasContract-to-PlutusContract convertor
- **DasContract.Blockchain.Plutus** contains all generator-related code

Both these projects have their respective NuGets published on nuget.org.

Each of these projects targets the .NET Standard 2.1 platform. This platform is widely supported, as stated by Microsofts’ documentation: *”Most general-purpose libraries should not need APIs outside of .NET Standard 2.0. .NET Standard 2.0 is supported by all modern platforms and is the recommended way to support multiple platforms with one target.”* [36]. The .NET Standard 2.1 will also be supported in all future .NET releases [36].

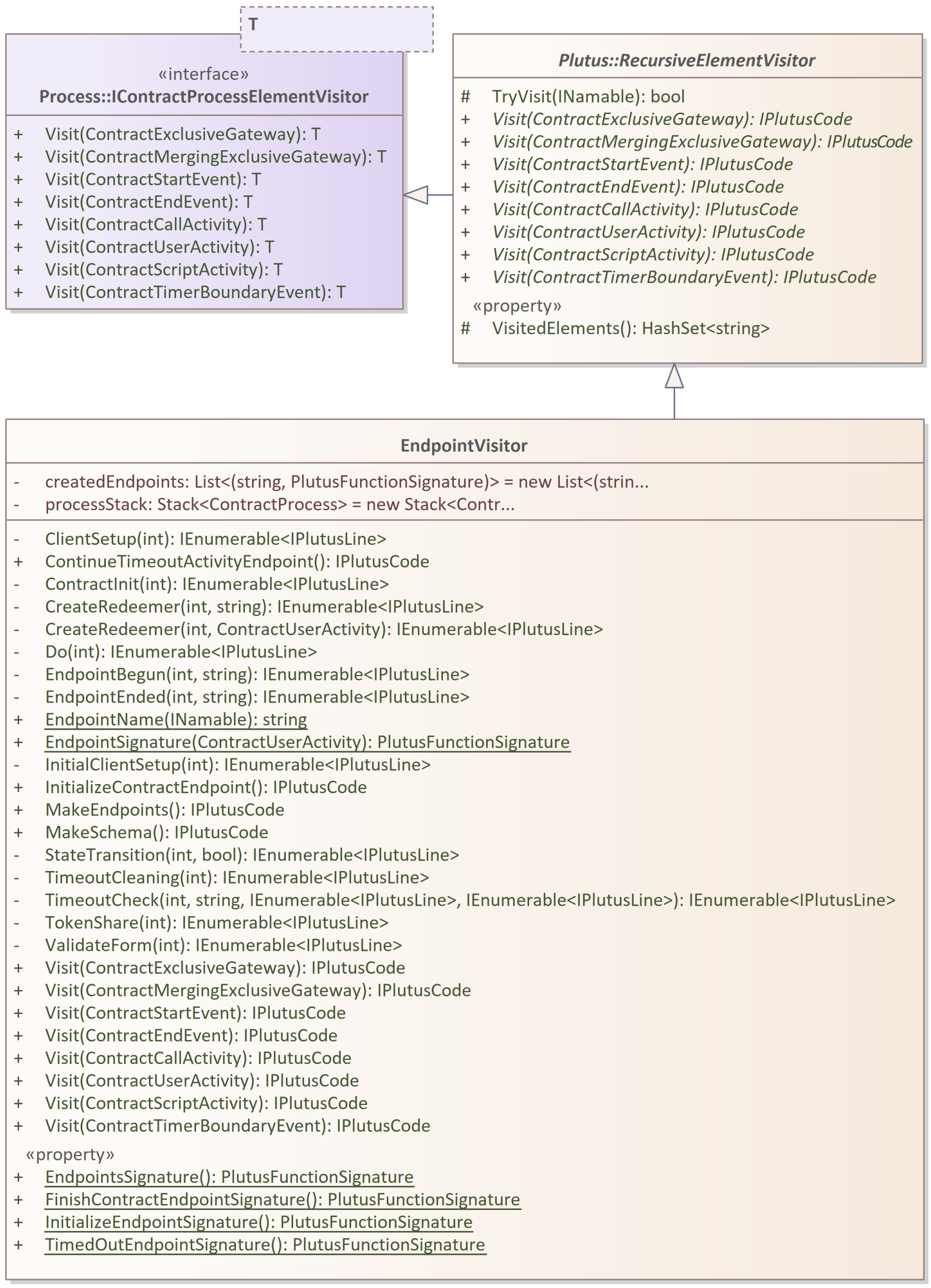

T
«interface»
Process::IContractProcessElementVisitor
+ Visit(ContractExclusiveGateway): T
+ Visit(ContractMergingExclusiveGateway): T
+ Visit(ContractStartEvent): T
+ Visit(ContractEndEvent): T
+ Visit(ContractCallActivity): T
+ Visit(ContractUserActivity): T
+ Visit(ContractScriptActivity): T
+ Visit(ContractTimerBoundaryEvent): T
Plutus::RecursiveElementVisitor
# TryVisit(INamable): bool
+ Visit(ContractExclusiveGateway): IPlutusCode
+ Visit(ContractMergingExclusiveGateway): IPlutusCode
+ Visit(ContractStartEvent): IPlutusCode
+ Visit(ContractEndEvent): IPlutusCode
+ Visit(ContractCallActivity): IPlutusCode
+ Visit(ContractUserActivity): IPlutusCode
+ Visit(ContractScriptActivity): IPlutusCode
+ Visit(ContractTimerBoundaryEvent): IPlutusCode
«property»
# VisitedElements(): HashSet<string>
EndpointVisitor
- createdEndpoints: List<(string, PlutusFunctionSignature)> = new List<(strin...
- processStack: Stack<ContractProcess> = new Stack<Contr...
- ClientSetup(int): IEnumerable<IPlutusLine>
+ ContinueTimeoutActivityEndpoint(): IPlutusCode
- ContractInit(int): IEnumerable<IPlutusLine>
- CreateRedeemer(int, string): IEnumerable<IPlutusLine>
- CreateRedeemer(int, ContractUserActivity): IEnumerable<IPlutusLine>
- Do(int): IEnumerable<IPlutusLine>
- EndpointBegun(int, string): IEnumerable<IPlutusLine>
- EndpointEnded(int, string): IEnumerable<IPlutusLine>
+ EndpointName(INamable): string
+ EndpointSignature(ContractUserActivity): PlutusFunctionSignature
- InitialClientSetup(int): IEnumerable<IPlutusLine>
+ InitializeContractEndpoint(): IPlutusCode
+ MakeEndpoints(): IPlutusCode
+ MakeSchema(): IPlutusCode
- StateTransition(int, bool): IEnumerable<IPlutusLine>
- TimeoutCleaning(int): IEnumerable<IPlutusLine>
- TimeoutCheck(int, string, IEnumerable<IPlutusLine>, IEnumerable<IPlutusLine>): IEnumerable<IPlutusLine>
- TokenShare(int): IEnumerable<IPlutusLine>
- ValidateForm(int): IEnumerable<IPlutusLine>
+ Visit(ContractExclusiveGateway): IPlutusCode
+ Visit(ContractMergingExclusiveGateway): IPlutusCode
+ Visit(ContractStartEvent): IPlutusCode
+ Visit(ContractEndEvent): IPlutusCode
+ Visit(ContractCallActivity): IPlutusCode
+ Visit(ContractUserActivity): IPlutusCode
+ Visit(ContractScriptActivity): IPlutusCode
+ Visit(ContractTimerBoundaryEvent): IPlutusCode
«property»
+ EndpointsSignature(): PlutusFunctionSignature
+ FinishContractEndpointSignature(): PlutusFunctionSignature
+ InitializeEndpointSignature(): PlutusFunctionSignature
+ TimedOutEndpointSignature(): PlutusFunctionSignature

Figure 3.12: A UML class diagram of the endpoints generator

3.6 Summary

Plutus contracts from the DasContract format are generated in two steps:

- DasContract is converted into PlutusContract, which is a more suitable format with a series of significant benefits
- PlutusContract is converted into Plutus code using Plutus generation tools

The PlutusContract data model is in many ways similar to DasContract, except few key differences. One significant difference is better, more concise OO design in data model properties and process elements. The other is an entirely different approach to a process model, where process elements are built recursively instead of linearly. This greatly simplifies the generation process, which is inherently recursive. There are other minor benefits, such as more straightforward changes to the generating process without the need for complex know-how.

The Plutus contract generator has a solid foundation of Plutus generation tools and interfaces. Fundamental models are recursive visitors capable of traversing entire processes and generating any code using a reliable and clean code. There are three of these visitors: Tx transitions visitor, NonTx transitions visitor, and endpoints visitor. These visitors are the most influential and impressive code of this thesis.

Codes for both steps are in their respective, highly compatible packages on the NuGet platform.

Testing of the generator is done primarily by systematic manual testing due to the evolving nature of the DasContract format, the testing difficulty of some areas, and the fact that this project is just a proof-of-concept, not a ready-made production product.

Chapter 4

Case study

Development and testing contracts often do not fully reflect reality. A real-life practical smart contract has been designed, implemented, and generated to explore the actual practicality of the Plutus smart contract generator.

The case study smart contract is a locking contract. The script (=contract) locks any funds from a person. These funds can only be retrieved if a defined deadline has passed by a predefined person – it may be the same person or a different one. A real-life application of this contract would be, for example, locking funds for ones' children to retrieve when they get older.

4.1 Users

There are two users in this contract:

- The one who locks the funds
- The one who retrieves the funds

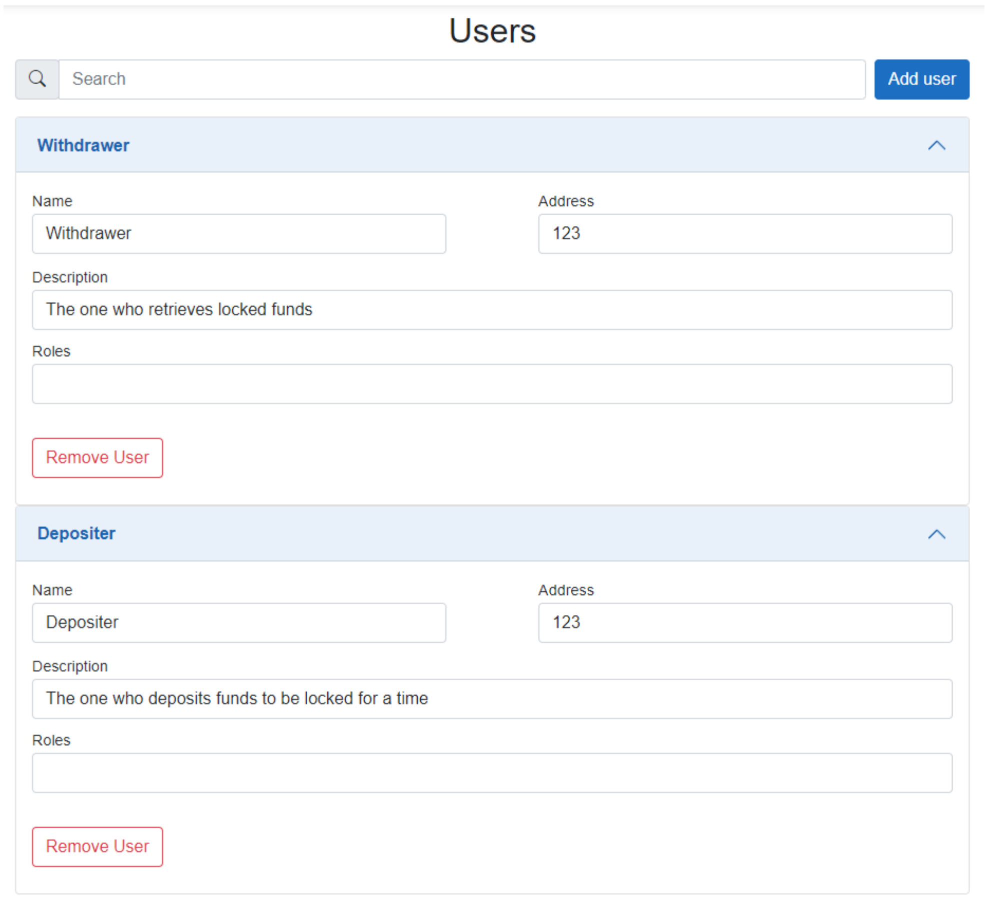
There is no need for roles in this contract.

Both personas can be the same; there are no limits for setting their wallet public keys.

4.2 Data model

The case-study contract does not require other properties than the following:

- `lockedLovelace` states the amount of Lovelace that will or has been locked
- `lockDeadline` states the deadline time in POSIX time



Withdrawer

Name	Address
Withdrawer	123
Description	
The one who retrieves locked funds	
Roles	

Depositer

Name	Address
Depositer	123
Description	
The one who deposits funds to be locked for a time	
Roles	

Figure 4.1: A snippet of a DasContract editor with the case-study contract – users

- `messages` is an array of informative messages or logs for potentially better user experience

4.3 Process

The case-study contract contains four processes.

The **setup subprocess** contains a user activity with a form for setting the locked amount and the deadline. It also includes a script task to log the successful finish of the setup.

The **successful withdrawal subprocess** contains a user task to execute the withdrawal of funds from the contract. It also contains a script task to log the successful withdrawal.

Listing 4.1 XML definition of the case-studies' data model

```
<DataTypes>
  <Entity Id="Root" Name="Datum" IsRootEntity="true">
    <Property Id="lockedLovelace"  Name="lockedLovelace"
      IsMandatory="true" PropertyType="Single"
      DataType="Int" />
    <Property Id="lockDeadline"  Name="lockDeadline"
      IsMandatory="true" PropertyType="Single"
      DataType="DateTime" />
    <Property Id="messages"  Name="messages"
      IsMandatory="true" PropertyType="Collection"
      DataType="String" />
  </Entity>
</DataTypes>
```

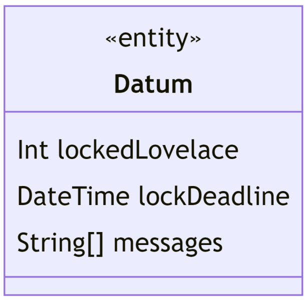


Figure 4.2: A snippet of a DasContract editor with the case-study contract – data model

The **failed unlock subprocess** records any unsuccessful withdrawal attempts.

The **main process** handles the essential structure and flow of the contract with the setup, unlock loop, timer condition, and eventual successful withdrawal.

4.4 Testing

An emulator program has been set up to test the case-study smart contract. The emulator is an in-memory simulation of a real blockchain.

The emulator monad sets up simulated wallets and invokes contracts endpoints. A heap of logs is then outputted to inform the user about inner workings and statuses.

The simulation test was a success, and everything ended up as it should have.

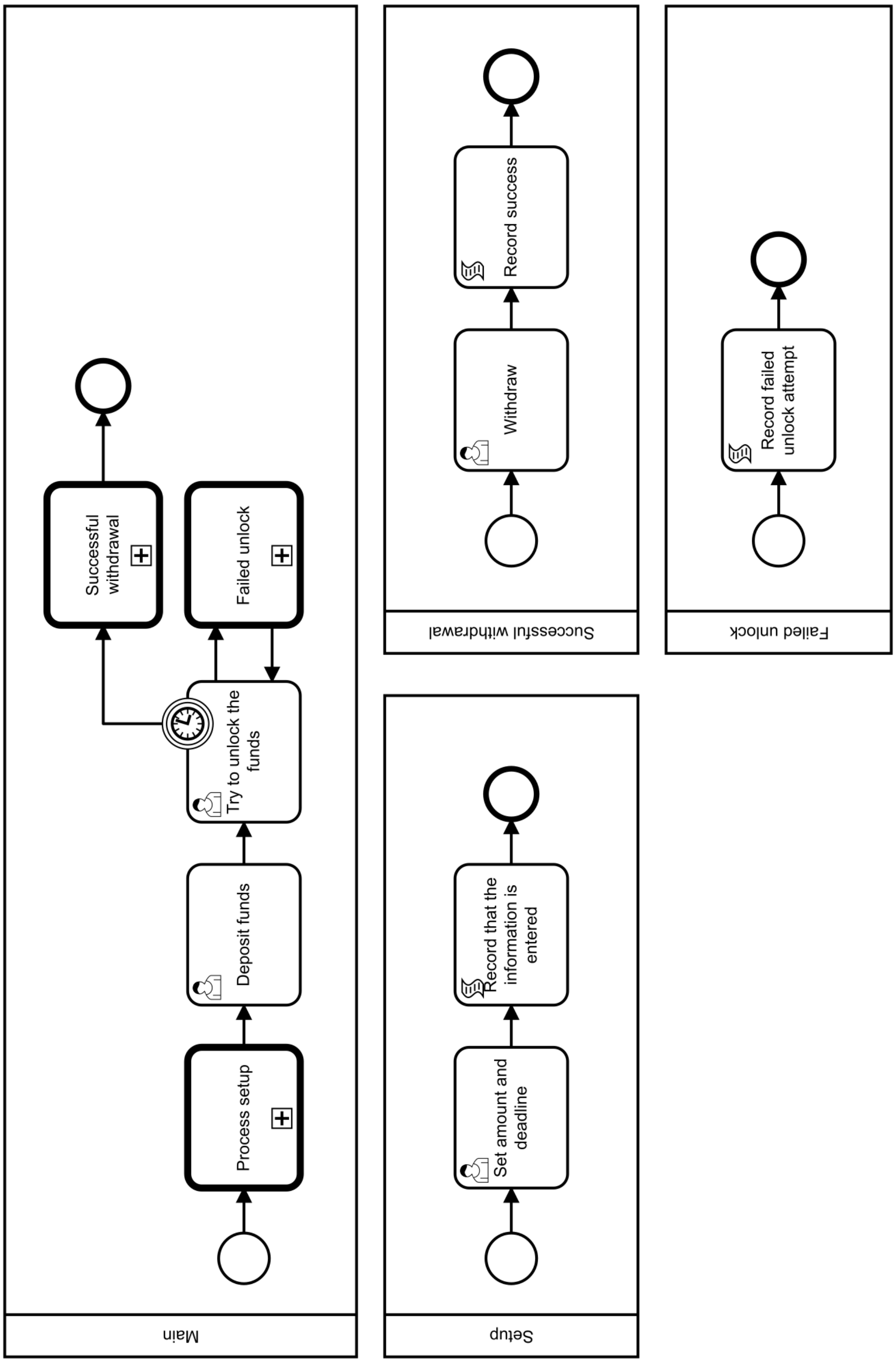


Figure 4.3: BPMN of the case-study contract

Listing 4.2 Case-study emulator trace test

```haskell
traceLockFunds :: EmulatorTrace ()
traceLockFunds = do
    Extras.logInfo $ show "Trace started"

    wallet1 <- activateContractWallet (knownWallet 1) endpoints
    let deadline = slotToEndPOSIXTime def 11

    --Initialize
    callEndpoint @"initializeContract" wallet1 ()
    threadToken <- getExistingThreadToken wallet1
    void $ Emulator.waitNSlots 3

    --Set funds and deadline
    callEndpoint @"setAmountAndDeadlineTask" wallet1
        (SetAmountAndDeadlineTaskForm {
            lockedLovelaceAmount = 10000000,
            lovelaceLockDeadline = deadline
        }, threadToken)
    void $ Emulator.waitNSlots 3

    ...

    --Withdraw
    callEndpoint @"withdrawTask" wallet1
        (WithdrawTaskForm, threadToken)
    void $ Emulator.waitNSlots 3

    -- Finish
    callEndpoint @"finishContract" wallet1 threadToken
    void $ Emulator.waitNSlots 3
```

4.5 Results and summary

A case-study smart contract has been designed, modeled, and implemented to test the Plutus smart contract generator on a real-life example.

The case-study contract is a locking contract capable of safekeeping locked funds for a set amount of time. The parameters are fully customizable, along with user wallets.

The case-study test ended up as a success, and a real-life applicable smart contract has been successfully completed and generated into the Plutus programming language.

Chapter 5

Plutus Generator state and future development

Most Plutus contract generator features were successfully implemented; however, a series of key features are currently missing. This chapter lists the implemented features and their functionality. It also lists the missing features, reasons for their exclusion, and proposes further research to implement them.

5.1 Implemented features

The proof-of-concept Plutus smart contract generator implemented the following DasContract features:

- Complete users and roles information generating and constraints
- Complete data model generating
- Full support for processes and subprocesses along with call activities
- XOR gateway branching
- Sequential multi-instances (loops)
- Script activities with custom datum transformation scripts
- User activities with form generating, custom form validation, scripts for value transfer, scripts for datum transformation, and custom constraints
- Timer boundary event with a preset date as a timeout

These features are not thoroughly tested and are not production-ready. However, they are stable with no known bugs and were capable of creating several fully working smart contracts, including a real-life case study.

5.2 Missing features

Since the Plutus smart contract generator is just a proof-of-concept, a series of features were not implemented. Some of them are not immediately possible due to the properties of the Cardano blockchain, and some require further extensive research or complicated implementations.

- Duration type timer for the timer boundary event has not been implemented since it is impossible to reliably retrieve the current time on the blockchain due to the nature of Cardano, which disables this feature, unless further research finds a workaround, for example, with oracles.
- Form data binding has not been implemented due to the difficulty of generating code since the immutable nature of Haskell records is not well suited for data-binding procedures.
- Loop property for multi-instances has not been implemented due to the complicated nature of compiling selector expressions and the nature of Haskell.
- Business rule tasks have not been implemented due to their overcomplicated nature. A DMN converter is now under research [29].
- Parallelism has not been implemented due to its complicated nature. Further research is required. Parallelism includes the parallel gateway and parallel multi-instances.

5.3 Future research and development

The proof-of-concept Plutus generator has a long journey with unknown obstacles ahead if it is to be a production-ready piece of software. However, there are immediate steps that should be taken into consideration for further research.

5.3.1 Bindings

Data bindings are a non-essential feature; however, they significantly ease the development complexity, which is what essentially DasContract does.

The first problem with data bindings is the immutable nature of Haskell/-Plutus data structures. Immutable structures are not capable of data binding. They can only be created with an updated value. There is an opportunity to generate function(s) to update the datum with bindings from a form.

If not for lists, dictionaries, and entity references, generating binding functions would be simple. These greatly complicate the generation process and the binding syntax. Thus, not only a complex interpreter for binding notation has to be implemented, but also a challenging generator for binding functions has to be implemented.

An interpreter for binding notation would also open doors for implementing loop properties for multi-instances.

To develop the data binding feature well and concisely, a research endeavor has to be undergone.

5.3.2 Testing

Testing is undoubtedly a crucial step toward excellent software. Currently, dozens of unit tests have been implemented for Plutus code generation tools, such as for instance generators, data structure generators, line generators (comments), etc. It is imperative to continue testing the application and make tests for the correctness of more essential and complicated modules.

5.3.2.1 Unit testing

An essential important step forward would be creating automated unit test for generating tools, especially transition, and endpoint generators.

There are dozens, even hundreds, of complicated situations which need to be tested. A more advanced testing framework needs to be created to allow solid unit-testing of these complicated generating modules. It is also crucial to make these unit tests well designed for future changes, as implementations or formatting of the resulting text may differ a bit – unit tests need to be simply fixable; otherwise they will become more of a burden than a help.

Unit testing should not be limited to the generator. More case-study contracts and tools should be made to test the generated Haskell code and its validity. This requires even more tools and redesigns to be made; however, it is critical.

Case-study contracts should also be tested on the Cardano testnet or even the production Cardano blockchain. Unfortunately, there are money to be involved (fees at minimum), and these tests should be conducted only, and only if at least unit tests thoroughly test the contracts.

5.3.3 Emulator generator

A first step in testing a contract is running an emulator and seeing the resulting logs, which explains what happens on the emulated blockchain.

These emulator programs can be automatically generated along with the smart contract, at least to some extent – the only problem is not knowing which paths to take in the process.

A fluent API Plutus smart contract emulator generator could be created to complement the Plutus generator and encourage testing. It is undoubtedly better to be able to quickly generate these emulators rather than painstakingly create them by hand and manually update them with every bit of change to the Contracts' process.

5.3.4 Parallelism

Parallelism has been left out of the current Plutus smart contract generator implementation, primarily due to the lack of research and complex implementation.

There are currently two threads of thought on the parallel implementation:

- ”Fake” multi-state parallelism
- ”True” parallelism

The ”fake” parallelism is based on keeping track of multiple states. Currently, only one state is stored, the current state, which is capable of transitioning to other states. Expanding this to an array of states could achieve a parallel behavior without extra significant effort. However, this is comparable to a single-threaded CPU running a multi-threaded program. From the programs' perspective, it runs multiple threads but in reality, does not, similar to the contract.

An example when this implementation will show its weaknesses is an immense amount of users trying to use the contract simultaneously. By nature, Cardano is capable of moving one contract by one Tx state in one slot, which is currently a timeframe of one second (maximum speed). If, for example, an election contract for millions of people was open, and two contract states would be parallelly available – a vote or a stop by an admin – the stop would be very hard to invoke since all the millions of people would constantly be voting, depleting the one slot. Additionally, the user experience for the voting contract may be horrid since people may be unable to vote due to the limited window.

The ”real” implementation would be truly forking the contract into multiple EUTXOs. However, this brings tons of issues, especially with synchronizing the data model. Currently, the DasContract format presumes that the data model is available everywhere, every time, always up to date [29]. Updating a single thread would mean updating all; however, this presents the same issue as described above.

One possible solution would be to break the DasContracts' rules and allow merging parallel gateway to contain merging scripts or create a smart datum merger. Another solution would be to enable the DasContract developer to choose the approach since the ”fake” parallelism is easier to implement for everyone and make the ”real” parallelism an extra feature.

5.3.5 PlutusContract module

The entire Plutus smart contract code is currently inside a `PlutusContract` module, with everything publicly visible.

More use-cases and more research are required to determine what is best to share and what not to share.

Additionally, the resulting code could be separated into multiple packages as not to be thousands of lines long with moderately complex contracts.

5.3.6 Contracts' efficiency

The efficiency of the Plutus smart contract generator is currently acceptable. It is mostly linear in complexity, and the generation process can take a few seconds max with hundreds of BPMN activities.

The efficiency of the generated Plutus code should be as high as possible since the complexity of the code directly affects transaction fees. Efficiencies such as cutting an unnecessary transition can improve execution speed. However, these improvements are complicated to map out and implement, especially since many situations can occur and the contracts' business logic must remain intact.

5.3.7 Yoroi Wallet Connector

It would be appropriate to assume that the average blockchain user cannot set up and submit their transactions for generated contracts. A proper UI environment with a connection to a users' wallet is one of the ways to provide smart contracts to everyday users. Of these wallet environments, one potentially suitable for this task is Yoroi.

"Yoroi is a light wallet for Cardano. It's simple, fast and secure. Yoroi is an Emurgo product, engineered by IOHK. And it follows best practices for software in the industry including a comprehensive security audit. [...] Yoroi looks to be a day to day wallet for a Cardano user." [37]

Future research with a generator that connects contracts to the Yoroi light wallet with the newly released Yoroi dApp Connector to Cardano would be a significant step toward providing DasContract services to everyday users [38].

5.4 Summary

The Plutus smart contract generator implemented most features, except parallelism, data binding, business rule tasks, duration timer, and other miscellaneous components. Unimplemented features either require ample time to implement or extensive research.

Parallelism should be among the earliest additions, as it enables more complex and powerful contracts. There are several options to approach parallelism, which need to be thoroughly researched and carefully implemented.

Testing is a vital part of development. The current generator contains a few unit tests; however, a full-scale comprehensive testing framework should be made to test the immense amount of possible situations. Smart contracts should have the utmost trust and always work as intended.

Conclusion

The objective of this thesis was to review the Cardano blockchain and its programming language Plutus and analyze, design, and implement a proof of concept generator from DasContract format into Plutus smart contract.

Analysis, design, and a proof of concept implementation of the Plutus smart contract generator have been successfully completed, and the fundamental structure of the code has been described. One case study of a funds locking smart contract has been created to demonstrate the usefulness and proper functioning of the generator.

Although the Plutus smart contract generator is working, it still lacks several key features of the DasContract format. Mainly, it is parallelism and data-binding. These features require further research and extensive programming effort. Additionally, exhaustive testing needs to be undergone to ensure the validity of the generators' code.

The DasContract format can now create contracts for the Ethereum network and, newly, the Cardano network. Some minor difficulties and blockchain-specific features with the DasContract have been found, though nothing that would make the creation of more generators impossible. Creating the second generator for a fundamentally different blockchain indicates that the DasContract format is solid and abstract enough to support a wide-scale amount of other blockchain networks.

Bibliography

1. SKOTNICA, Marek; PERGL, Robert. Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts. In: AVEIRO, David; GUIZZARDI, Giancarlo; BORBINHA, José (eds.). *Advances in Enterprise Engineering XIII*. Lisbon, Portugal: Springer International Publishing, 2019, pp. 149–166. ISBN 978-3-030-37932-2. Available from DOI: `10.1007/978-3-030-37933-9_10`.
2. DROZDÍK, Martin. Open-source prostředí pro návrh právních procesů za použití frameworku Blazor. *2019* [online]. 2020, vol. 2019 [visited on 2022-02-07]. Available from: `https://dspace.cvut.cz/handle/10467/88271`. Accepted: 2020-06-19T22:51:49Z Publisher: České vysoké učení technické v Praze. Vypočetní a informační centrum.
3. *Cardano is a decentralized public blockchain and cryptocurrency project and is fully open source.* [Cardano] [online] [visited on 2022-03-21]. Available from: `https://cardano.org/`.
4. *BI-BEZ Lecture #6 - RSA, kryptografie s veřejným klíčem, DSA, ElGamalův algoritmus.* Ve spol. s ING. LÓRENCZ, Róbert prof. 2019. Available also from: `https://courses.fit.cvut.cz/BI-BEZ/lectures.html`.
5. *BI-BEZ Lecture #5 - Hašovací funkce, MD5, SHA-x, HMAC.* Ve spol. s ING. LÓRENCZ, Róbert prof. 2019. Available also from: `https://courses.fit.cvut.cz/BI-BEZ/lectures.html`.
6. LARS BRÜNJES. *Plutus Pioneer Program - Iteration #2 - Lecture #7* [online]. 2021 [visited on 2022-02-08]. Available from: `https://www.youtube.com/watch?v=uwZ903Zd0DU`.
7. NAKAMOTO, Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System [online]. 2019 [visited on 2020-04-07]. Available from: `http://www.bitcoin.org/bitcoin.pdf`.

8. IOHK. *Cardano Docs* [Cardano] [online] [visited on 2022-02-07]. Available from: `https://docs.cardano.org/`.

9. *Ethereum development documentation* [ethereum.org] [online]. 2021 [visited on 2022-02-08]. Available from: `https://ethereum.org`.

10. LARS BRÜNJES. *Plutus Pioneer Program - Iteration #2 - Lecture #1* [online]. 2021 [visited on 2022-02-08]. Available from: `https://www.youtube.com/watch?v=_zr3W8cgzIQ`.

11. *Understanding the Extended UTXO model* [online] [visited on 2022-02-08]. Available from: `https://docs.cardano.org/10-plutus/02-eutxo-explainer/`.

12. *HaskellWiki* [online] [visited on 2022-03-21]. Available from: `https://wiki.haskell.org/Haskell`.

13. HUDAK, Paul. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys* [online]. 1989, vol. 21, no. 3, pp. 359–411 [visited on 2022-03-21]. ISSN 0360-0300, ISSN 1557-7341. Available from DOI: `10.1145/72551.72554`.

14. *Haskell Language* [online] [visited on 2022-03-21]. Available from: `https://www.haskell.org/`.

15. LARS BRÜNJES. *Plutus Pioneer Program - Iteration #2 - Lecture #4* [online]. 2021 [visited on 2022-02-08]. Available from: `https://www.youtube.com/watch?v=g4lvA14I-Jg`.

16. *Why use Cardano?* [Online] [visited on 2022-03-19]. Available from: `https://docs.cardano.org/03-new-to-cardano/03-why-use-cardano/`.

17. *Input Output* [IOHK] [online] [visited on 2022-03-19]. Available from: `https://iohk.io/en/technology/`.

18. *Ouroboros* [online] [visited on 2022-03-19]. Available from: `https://cardano.org/ouroboros/#proof-of-stake`.

19. *Cardano nodes* [online] [visited on 2022-03-19]. Available from: `https://docs.cardano.org/03-new-to-cardano/05-cardano-nodes/`.

20. *About Cardano - Understanding Consensus* [online] [visited on 2022-03-19]. Available from: `https://cardano-foundation.gitbook.io/stake-pool-course/lessons/introduction/about-cardano#slot-leader-election`.

21. *Learn about native tokens* [online] [visited on 2022-03-20]. Available from: `https://docs.cardano.org/native-tokens/learn`.

22. LARS BRÜNJES. *Plutus Pioneer Program - Iteration #2 - Lecture #5* [online]. 2021 [visited on 2022-02-08]. Available from: `https://www.youtube.com/watch?v=SsaVjSsPPcg`.

23. *Learn about Plutus* [online] [visited on 2022-03-21]. Available from: `https://docs.cardano.org/10-plutus/01-learn-about-plutus/`.

24. *Transaction costs and determinism* [online] [visited on 2022-03-21]. Available from: `https://docs.cardano.org/10-plutus/11-transaction-costs-determinism/`.

25. LARS BRÜNJES. *Plutus Pioneer Program - Iteration #2 - Lecture #2* [online]. 2021 [visited on 2022-02-08]. Available from: `https://www.youtube.com/watch?v=sN3BIa3GAOc`.

26. R, Manoj P. *Most common smart contract bugs of 2020* [Solidified] [online]. 2020-11-30 [visited on 2022-03-21]. Available from: `https://medium.com/solidified/most-common-smart-contract-bugs-of-2020-c1edfe9340ac`.

27. SKOTNICA, Marek; FRAIT, Jan; KLICPERA, Jan; DROZDÍK, Martin; ONDŘEJ, Šelder. *CCMiResearch/DasContract* [GitHub] [online] [visited on 2022-02-19]. Available from: `https://github.com/CCMiResearch/DasContract`.

28. FRAIT, Jan. Generating Ethereum Smart Contracts from DasContract Language. *2020* [online]. [N.d.] [visited on 2022-02-19]. Available from: `https://dspace.cvut.cz/handle/10467/90034`.

29. SKOTNICA, Marek. *Personal conversation: DasContract language.* Praha, 2022.

30. *Business Process Model & Notation™ (BPMN™) — Object Management Group* [online] [visited on 2022-03-17]. Available from: `https://www.omg.org/bpmn/`.

31. ANČINEC, Petr. Domain-Specific Languages for Off-chain UI in Decentralized Applications [online]. 2021 [visited on 2022-02-19]. Available from: `https://dspace.cvut.cz/handle/10467/94542`. Accepted: 2021-06-03T22:52:43Z Publisher: České vysoké učení technické v Praze. Vypočetní a informační centrum.

32. *Decision Model and Notation™ (DMN™) — Object Management Group* [online] [visited on 2022-03-18]. Available from: `https://www.omg.org/dmn/`.

33. *Finite State Machines — Brilliant Math & Science Wiki* [online] [visited on 2022-03-22]. Available from: `https://brilliant.org/wiki/finite-state-machines/`.

34. *Call stack - MDN Web Docs Glossary: Definitions of Web-related terms — MDN* [online] [visited on 2022-03-22]. Available from: `https://developer.mozilla.org/en-US/docs/Glossary/Call_stack`.

35. *Making Our Own Types and Typeclasses* [online] [visited on 2022-03-25]. Available from: `http://learnyouahaskell.com/making-our-own-types-and-typeclasses`.
36. GEWARREN. *.NET Standard* [online] [visited on 2022-03-27]. Available from: `https://docs.microsoft.com/en-us/dotnet/standard/net-standard`.
37. *Yoroi - Light Wallet for Cardano* [online] [visited on 2022-04-11]. Available from: `https://yoroi-wallet.com/`.
38. *Yoroi Wallet's dApp Connector Now Available for Cardano Ecosystem* [Emurgo] [online] [visited on 2022-04-11]. Available from: `https://emurgo.io/blog/yoroi-wallets-dapp-connector-now-available-for-cardano-ecosystem`.

Appendix A

Acronyms

BPMN	Business Process Model and Notation.
DMN	Decision Model and Notation.
EUTXO	Extended Unspent Transaction Output.
FT	Fungible Token.
I/O	Input/Output.
LIFO	Last In – First Out.
NFT	Non-Fungible Token.
NonTx	Non-Transactional.
OO	Object-Oriented.
Tx	Transactional.
UI	User Interface.
UML	Unified Modeling Language.
UTXO	Unspent Transaction Output.
XML	Extensible Markup Language.

Appendix **B**

Contents of enclosed CD

```
readme.txt ....................... the file with CD contents description
src ....................................... the directory of source codes
  case-study ............................. the case study source files
  application ............................... 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
```