TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technology

Chair of Software Technology

Bachelor Thesis

Contract to Contract Calls for Financial Applications in Algorand

Vasil Petrov

Born on: April 8, 1999 in Burgas, Bulgaria
Matriculation number: 4818111
Matriculation year: 2018

to achieve the academic degree

Bachelor of Science (B.Sc.)

Supervising professor
Prof. Dr. Uwe Aßmann

Submitted on: August 17, 2022

Contents

Abstract

here goes the abstract, I am going to write it when I am done with the rest, see you then ;)

1 Introduction

1.1 Motivation

Blockchain technology has the potential to revolutionize the financial world. Its industrial adoption is growing at a rapid rate mainly due to the rising popularity of cryptocurrency. Big names in the banking sector such as Bank of America, Agricultural Bank of China, HSBC, BNP Paribas, and many more have already invested a lot of capital into researching and experimenting with blockchain solutions for their financial services with the hope of reducing processing costs and time.[15]

In simple terms, blockchain represents a decentralized immutable ledger where transactions and assets can be securely stored and processed. Through cryptography and other mechanisms used by this technology, the need for a trusted central authority as a middleman in the transaction between two individuals is redundant. This could open the doors to the global financial market for a lot of people that don't have access to modern banking services.[17] Fintech startups are quickly trying to capture this new market by developing financial applications on various blockchain platforms such as Algorand.

Algorand is a cutting-edge blockchain technology created by Silvio Micali, a Turing Award winner, and his team of leading scientists. On it, developers can build complex applications with the help of smart contracts which can be interpreted as programs that are stored and run on the distributed ledger. Once a smart contract is written and deployed on the blockchain, it becomes immutable and can execute logic automatically when certain conditions are met. This allows the different sides involved in the contract to have trust in the outcome without the need for a neutral party as a guarantor. However, the immutability of the smart contracts can sometimes be bad news for the software development process, because quite often certain parts of the program need to be optimized or rewritten. Luckily there are ways to get around this drawback by using upgradable smart contracts. The idea is that all transactions and values that have already happened on-chain will be visible and immutable on the ledger until it exists. Nevertheless, we can still swap the logic of a smart contract with a new upgraded one. We will examine how this can be achieved on Algorand and other famous blockchain protocols later in the thesis.

As suggested in this paper[18] by T. Mattis and R. Hirschfeld, an analogy could be made between smart contracts and objects in object-oriented programming. Each contract has its own *state* and *methods* and plays its specific part in the application. Object-oriented programming (OOP) is a paradigm that is used for designing modular, reusable software. With this approach, a complex problem could be solved by dividing it into a set of subproblems among different classes, making the development process easier and more intuitive. In the blockchain world, this could be achieved using contract composability which refers to com-

bining multiple smart contracts, where each solves a certain aspect of the global problem, achieving again a modular structure. The way to achieve composability is by contract to contract calling. This allows one smart contract to interact and execute transactions to another contract on its own. Thus, with just one transaction sent by the client of the application, a chain of complex logic and communication between different contracts can be triggered. Furthermore, this enables developers to divide the system into different logical sections that can be called by users according to their role and authorization. In other words, we can apply a role-based approach to the application.

As mentioned in [23], the idea behind role-oriented programming is that the same object (entity) in different contexts could play different roles. Each role serves a specific purpose and can overwrite the behavior of the object allowing it to adapt to a certain context dynamically. For example, a customer of a bank can be at the same time a borrower and depositor. There is no point to model borrower and depositor as separate objects, but instead, as roles that can be added or dropped dynamically by customer objects. This way we can achieve a more accurate and natural representation of the real world in our program.

Combining a role-based programming approach with blockchain could be a very powerful concept. However, there still exists a gap in research for financial applications using blockchain from an academic perspective and this could hurt the rate of adoption of this relatively new technology by the banking sector. For this reason, the main objective of this thesis is to investigate whether a financial role-based application can collaborate with the Algorand blockchain. If possible, the development process should be documented and practically demonstrated with a simple Proof of Concept (PoC).

The primary task of this thesis will be to answer the following research questions:

1. How to make the Algorand blockchain interact and be a part of a role-based application?
2. How contract to contract calling and upgradable smart contracts help with the development of complex applications on the Algorand blockchain?
3. What are the drawbacks and advantages of developing role-based applications using the programming language of Algorand?

1.2 Goal and objectives

To answer the research questions asked in Motivation, a simple banking application can be planned as a PoC. The key part is that all of the important data and transactions that take place in this bank should be recorded on the Algorand blockchain, where they will be securely stored and immutable. Furthermore, most of the bank functionality such as opening a bank account, depositing, withdrawing, and transferring funds should be implemented on the blockchain as well via smart contracts. This will demonstrate how programs on the ledger could communicate and cooperate to build more complex business logic. The app will implement also a role-oriented approach to help us understand whether a smart contract can react and execute different logic depending on the role that is calling it. The main objective of the PoC is to demonstrate whether a role-based application written with a general-purpose language like Python can collaborate with the Algorand blockchain. The results of this research should help the banking sector to see the process and benefits of using blockchain as a technology for their financial applications.

1.3 Structure of the thesis

This thesis is divided into six chapters.

First and foremost, the topic of the thesis is laid out together with the goals and objectives in the Introduction section.

In Related work, various research papers conneted to the topic of the thesis will be presented, further specifying the direction of the study and explaining why it should be conducted

The reader should be familiar with some basic concepts to better grasp and follow the research process, this is done in the Background chapter. There, concepts like role-based programming, blockchain, smart contracts and the Algorand protocol will be explained in more detail.

In section four comes the practical part. A detailed description supported by code snippets will illustrate the development process and how the bank application part by part is being created.

The test results regarding the functionality of the developed app are shown in the Evaluation section of this thesis. An argument will be presented on how these results provide an answer to the research questions.

In the Discussion chapter, the pros and cons of developing applications on the Algorand blockchain will be considered. The arguments will also be supported by performance tests. Reference and comparison to another popular blockchain platform, namely Ethereum, will also be made here.

Finally, the Conclusion section presents perspectives for the extension of the presented research and app as well as a summary of the researched questions and process.

I will update this part after I am done

2 Related work

There is a good amount of academic articles regarding role-based programming. In the scientific paper "Refactoring to Role Objects"[] by F. Steimann and F. Stolz, it is described how the object-oriented paradigm has its limitations when it comes to modeling systems that are represented by roles. The idea of the role-based approach is that just as we as humans perform different roles (duties) throughout our daily lives, we should also be able to dynamically add and remove roles to a class object. The logical entity will remain the same, but its behavior will change based on different contexts. As a way to implement the concept of roles in an application, the authors have mentioned the Role Object Pattern (ROP). This pattern was first proposed by D. Bäumer et al. in the paper "The Role Object Pattern"[]. The idea is to model every specific context of an object in a separate role object that can be dynamically added or removed at runtime. As stated in the paper, this pattern achieves the role-based approach while avoiding 'the combinatorial explosion of classes' caused by implementing the concept via OOP inheritance. Low coupling of the different contexts is also achieved and every role can be changed independently without affecting the others. These are some of the benefits of this pattern. There are also other approaches and methods to model the role-based approach. L. Schütze and J. Castrillon in their work "Analyzing State-of-the-Art Role-based Programming Languages"[] have analyzed and compared 4 different approaches to implementing the concept of roles in a system. One of these is again the Role Object Pattern. They compare the performance of each approach based on scalability and memory management. In the end, the results showed that ROP is by far the best in terms of realizing roles in an OOP environment. The authors have also noted that nowadays more and more sectors require adaptive and context-aware software. In their opinion, the role-based approach is a good solution for this increasing need.

From the papers presented so far, there is clearly an interest in role-based programming as an approach to innovate and supplement the object-oriented paradigm. However, I couldn't find any academic papers that try to demonstrate role-based programming using blockchain technology.

TODO - add a paper for decentralized applications and explain that software development now on blockchain is possible. However there is a lack of research regardin role-based applications. Finish up tomorrow

3 Background

In this chapter, the necessary basic knowledge of different concepts is provided which is key for understanding the rest of the thesis. We begin with a more in-depth definition of blockchain technology and what advantages it offers. After that, smart contracts are going to be discussed with an explanation of features such as contract to contract calling and upgradeable contracts. Then we will continue with a description of the Algorand blockchain and its perks. Finally, we will end with a more detailed explanation of the role-based programming approach.

3.1 Blockchain technology

Many consider blockchain as a revolutionary technology that is the next big thing. It is even compared in the potential to the early years of the internet. However, many do not know that its basic idea and fundamentals predate the emergence of cryptocurrencies.

3.1.1 History and Structure

David Chaum in his dissertation from 1982 called "Computer Systems Established, Maintained, and Trusted by Mutually Suspicious Groups"[8] is one of the first to suggest a protocol that resembles blockchain as we know it today. He proposed the implementation of a decentralized system that can be maintained and trusted by achieving mutual consensus between its participants. Almost a decade later S. Haber and W. Scott Stornetta proposed in their work[14] a time-stamping mechanism for digital documents. The process involves a one-way hash function where the hashes of the different time-stamped documents are linked together, forming a tamper-proof cryptographically secured chain. Based on these and some other early works, Satoshi Nakamoto published a whitepaper titled "Bitcoin: A Peer-to-Peer Electronic Cash System"[20], where blockchain was introduced as a public distributed ledger for bitcoin payment transactions. From this point on, the rising popularity of this technology and cryptocurrencies began.

As explained by A. Panwar and V. Bhatnagar in [21], blockchain is just a type of distributed ledger technology (DLT). This is important to note because many people use both terms interchangeably which is technically wrong. DLT is a decentralized database distributed as copies across multiple computer nodes connected via a peer-to-peer (P2P) network. If a modification occurs, then all copies of the ledger get autonomously updated and synchronized. With the things said so far, it sounds exactly like the definition of blockchain. However, the key difference is that DLT doesn't necessarily need to represent its data in a block

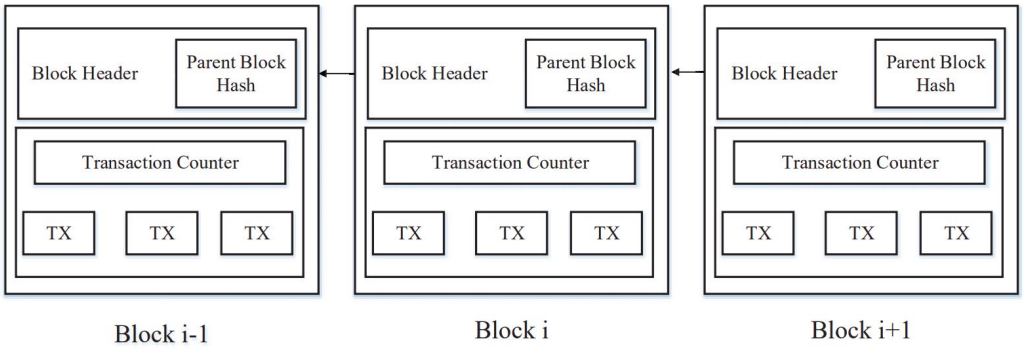

Figure 3.1: An example of a basic blockchain structure (Figure 1 in [30])

structure, but in any type of structure. Furthermore, DLT doesn't necessarily store data in a specific sequence. Many blockchains also use some kind of native currency/token on the network, used to pay fees for interacting with the network and also as an exchange value between different participants. It is also given as a reward for the people who keep the blockchain alive and add/validate new blocks to the chain. However, not all forms of DLT require a token or a currency to keep the system going. These are just some of the key differences between the two technologies.

As described by Z. Zheng et al.[30], blockchain can be viewed as a sequence of blocks that are linked together in a chronological order using cryptography. As can be seen on Figure 3.1, each block is composed of a *block header* and a *block body*. The block header holds the hash value that points to the previous block (also called a parent block) and a timestamp that shows when the block was validated and added to the chain. In each block body, a certain number of transactions is being stored depending on the size of the block and each transaction. Any attempt to modify some information in an existing block will affect its hash address and all others before and after it because everything is connected. The participants will compare the change with their current copy of the ledger and immediately notice the change and reject it. This is why blockchain is considered immutable and tamper-proof.

As further mentioned in [30], asymmetric cryptography (public-key cryptography) is one of the key components of blockchain technology. Each participant in the network owns a pair composed of a private key and a public key. The private key is used to verify transactions and prove ownership of a blockchain address and the public key is used for the unique account address, allowing users to receive cryptocurrencies or other digital assets.

3.1.2 Types of blockchain

Blockchain networks can be divided into 3 categories: public, private and consortium.[16]

Public blockchains are permissionless, meaning that they are open networks, where anyone can participate and read/write to the ledger without needing the approval of any trusted authority. The transactions are completely transparent and accessible for anyone to see, but in most cases they are anonymous and you can't directly link them to the person behind them. Here we see anonymity and transparency, qualities for which blockchain is liked and which are absent in the standard financial system.

Private blockchains on the other hand require permission to join them. They are more suitable for organizations and companies which want to benefit from its implementation but don't want everyone to have control over the network. Normally, it is again a centralized structure, where only some participants have the right to verify and add new data to the

chain.

In consortium blockchains, there is more than one company/organization in charge to decide who has what rights on the blockchain. It is best suited for organizational collaboration. It can be viewed as a hybrid between public and private blockchain because the power is not only in one central structure, but at the same time, not everyone can participate and interact with the network.

From this moment forth, the word blockchain will refer to the public version of the network, because this thesis will revolve around Algorand, which is a permissionless public blockchain.

3.1.3 Consensus mechanisms

Based on the provided knowledge so far, blockchain is a decentralized system available for everyone to take part in it. Everyone has an updated and synchronized copy of the ledger. The control of the network is not in the hands of some central entity but all participants in the network. However, what guarantees that the copy of each node will be up to date and correct? Who validates and adds new transactions/data to the chain? How is it verified that these new transactions are valid at all? Here comes the important role of a consensus mechanism. The most used ones are Proof of Work (PoW) and Proof of Stake (PoS), which are going to be briefly explained in the following paragraphs.

PoW was the first consensus mechanism ever for blockchain. It was introduced in the whitepaper for Bitcoin[20], where Nakamoto argued that this will be the mechanism to keep the distributed ledger secure and consistent while being decentralized and without a central authority to regulate. As explained in the research article "On the Security and Performance of Proof of Work Blockchains"[12] by A. Gervais et al., the PoW principle of selecting a node to add a new block to the chain is by propagating to the network a cryptographic puzzle that needs to be solved. Every node also called a miner, is in direct competition with the others and spends computational resources and time to find the solution. Once an answer is found, it gets propagated to the network together with the new block and the other nodes verify if they are valid or not. If yes, the new block gets appended to the chain and all participants update their ledgers. If not, the block will be rejected and a new proposal will be expected. This process is repeated and thus the chain is kept consistent, valid and secure.

PoS has a different approach to reaching a consensus. As described by F.Saleh in [22], PoS chooses the proposer (forger) of a new block and the validators based on their stake or in other words on the proportion of native coins they are holding. The protocol chooses a random coin each round from the supply and the node to whom it belongs will be entitled to propose a new block on the chain. This is one of the reasons why PoS is considered a more elegant mechanism than PoW and its use in new blockchain projects is increasing.[22] They are better in terms of scalability and require much fewer resources such as electricity or investment in equipment to participate.

3.1.4 Summary of blockchain's characteristics

As a summary of the previous three subsections, blockchain is an immutable, decentralized, distributed, transparent and secure network that is maintained and managed by all its participants that run its software. These participants are called nodes and each has its copy of the ledger that holds all of the transactions and information that are so far written on the chain. This supports the claim of distributed and transparent nature of the system. Furthermore, the authority and control belong to all the nodes, not to a single central structure. With the help of a consensus mechanism, the participants can collaborate without problems in adding new blocks to the chain and maintaining its integrity and consistency. This

achieves decentralization and security of the network and a transaction is accepted only if the majority of nodes agreed that it is valid. Thanks to the cryptography used in the protocol, already existing records on the chain can't be edited or deleted. This guarantees the immutability of the transactions made on the blockchain.

3.2 Smart contracts

Companies from various fields are looking for ways to optimize the quality and efficiency of their services and products through blockchain. Every one of them has some arrangement with their customers. For instance, if you pay an X amount of money, you get some service Y in return. If you don't like the quality of the service or product, then you can complain and get your money back. This is a simple example, but any kind of business operates more or less on the same principle. To recreate this logic in blockchain, we need the concept of smart contracts.

3.2.1 History and meaning

The term *smart contract* was coined for the first time by American cryptographer Nick Szabo back in the 90s.[24] That was long before the Bitcoin whitepaper[20] and the emergence of blockchain as a technology. It was described by Szabo as a digital form of a contract between two or more parties that can give you automatically an outcome as soon as some predetermined conditions are met. The main idea of this tool was to automate and optimize the workflow of business processes and eliminate the role of intermediaries that act as trust between the parties. The thing that would monitor the proper execution of the terms of the agreements and build trust between the different parties is the code itself. This definition is still relevant today, but it took nearly two decades before blockchain put smart contracts into practice.

As explained in the Ethereum whitepaper[4] by Vitalik Buterin, the scripting language for the Bitcoin protocol has its boundaries. It has limited programmability for the sake of reducing potential vulnerabilities and increasing the security of the decentralized currency. For instance, loops are not allowed in order to evade infinite loops. This makes the scripting language of Bitcoin non-Turing complete. Another problem mentioned by Buterin is the lack of state, meaning that the Bitcoin blockchain can't distinguish other states of a transaction except for *spent* and *unspent*, making it impossible to implement multi-stage smart contracts, which require more options and intermediate states to replicate real-world conditions and agreements. In other words, even after the creation of Bitcoin and the emergence of blockchain as a protocol, smart contracts still couldn't be implemented. A few more years had to pass to turn the cooperation between the two concepts into reality.

According to [26], Ethereum was the first blockchain to enable smart contract programming and remains the most widespread platform for developing decentralized applications. This is done via the Ethereum virtual machine (EVM) which is Turing-complete. Now smart contracts can exist as programs on the blockchain, read/write values on it, execute transactions, exchange assets or call and collaborate with other contracts to create complex application logic running on the ledger. Because of the distributed nature and immutability of the blockchain, it could be guaranteed that no one can edit or delete an already deployed smart contract and that if the specific conditions are met, a certain outcome will happen. Furthermore, each contract can be represented as a regular account that can hold its own assets/funds and has its own unique address on the network. Since the appearance of Ethereum, smart contracts have been an essential component of blockchain technology and many companies are taking advantage of their qualities. Algorand also uses smart con-

tracts and allows the development of applications on its platform, but more details on this are provided in the Algorand subsection of this chapter.

3.2.2 Upgradable smart contracts

As stated here [1], once a smart contract is deployed on the blockchain, it becomes immutable. If a new condition needs to be added or a security issue to be fixed to the existing code, then a new version of the contract should be deployed. This is not at all convenient for application development because it requires very careful pre-consideration and is also expensive to maintain. If you swap a smart contract with a newer version, then you have to inform all other contracts and users that interact with it about its new address and functionality. Otherwise, the old version will continue to be called and the update will be in vain. This drawback could prove to be a big barrier for companies looking to adopt blockchain solutions to their business. Luckily, people from academia and industry have come up with some ways around the immutability problem.

As mentioned in "Evaluating Upgradable Smart Contract"[3] by Van Cuong Bui et al., smart contracts on Ethereum can't be modified once deployed, but by using design patterns we can work around the code's upgradeability issue. Each contract consists of logic (functions) and state (data). The proxy pattern uses the idea of splitting the smart contract into one responsible for the data storage and the other for the logic. As shown in Figure 3.2 the proxy contract is the one taking care of the data and all the storage changes are happening there. The proxy also stores in its state the address of the current logic contract. As described in [3], when a caller (user) calls the proxy, it delegates the call to the current logic contract where all of the functionality is. Because we have delegation, the functions being executed in the logic contract will be in the context of the proxy, meaning they will use the proxy's data storage. Then the logic contract will return a result to the original caller again via the proxy. As highlighted in the figure below, if we want to add some new functionality, we just deploy a new logic contract and tell the proxy its address. From this moment forward, the proxy will delegate all external calls to the new version. With the proxy pattern upgradability of smart contracts is achieved without hurting the immutability factor. No contract gets deleted or modified, just the logic contract is swapped with a new one and the proxy starts to delegate calls from users or other smart contracts to the newer version instead of the older one. This is also a lot easier to maintain because now only the proxy needs to be informed of the new address.

It is important to remark that on Algorand these kind of patterns are most of the time not required. Contrary to Ethereum, smart contracts can be updated directly, and the right to update is controlled by the application itself. However, only the logic part of a contract can be updated without the need to deploy a new version. The amount of data storage remains immutable. In the end, we achieve the same result as by using design patterns. More on the matter will be explained in the Algorand subsection of this chapter and Implementation.

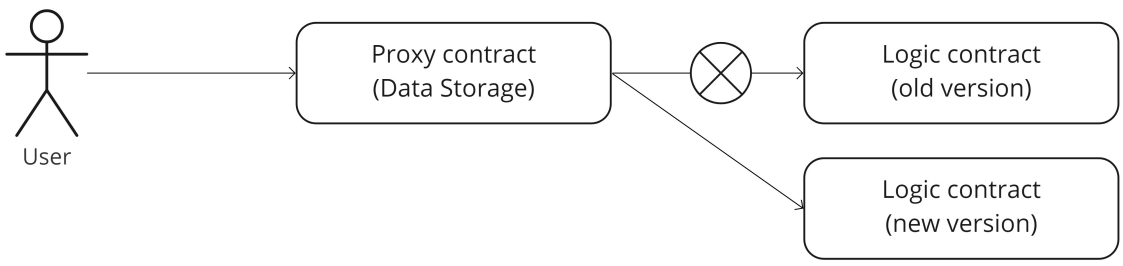

Figure 3.2: Proxy Pattern

3.2.3 Contract to contract calls

For blockchain to be adopted by large companies, then writing complex applications on it must be possible. We have found that with smart contracts this can be achieved. Nonetheless, we cannot afford to write all the business logic in just one contract. First, it is against the good practices of software engineering. If we want the code to be easier to maintain, we need modularity and interoperability between the different building blocks of the program. Second, smart contracts are also limited in their size. On Ethereum for example, the maximum size of a contract is limited to 24KB [1], on Algorand it is 8KB [2]. These limitations further reduce the possibility of writing large programs in just one contract.

This is why the possibility of communication between smart contracts at runtime is crucial for developing complex applications that follow good practices. As described in this article[29][3], Algorand allows a smart contract to call, share information and even create other smart contracts. This enables developers to build powerful and versatile decentralized applications on the Algorand blockchain.

3.3 Algorand

3.3.1 Goals and advantages

Algorand is one of the most advanced open source permissionless public blockchains currently in the field. Founded in 2017 by MIT professor and Turing Award winner Silvio Micali, the goal of the Algorand protocol is to solve the blockchain trilemma or in other words, to be a truly decentralized, secure, and scalable system.[10] Micali and his colleagues saw that most blockchains like Bitcoin or Ethereum have weaknesses in fulfilling these 3 requirements simultaneously because of their consensus mechanism PoW and because of their overall architecture.[9] As noted in [11], some of the flaws of using PoW are that it requires too many resources and time to reach consensus, short-lived forks are a possibility and there is some degree of centralization because of the big mining pools, where miners combine their computational efforts to increase chances of proposing new blocks and get rewards for it. As explained in [25], forks happen when the blockchain splits into alternative paths, often happening because of disagreement between miners about which block should be the next added to the chain. Eventually, all of the miners go back to the longest version of the chain and the alternative ones get abandoned. As mentioned in [13], this is why users of the Bitcoin blockchain have to wait up to an hour to be certain that their transactions were confirmed and added to the main chain and not on a forked version that will be discredited further down the line. Forks are the cause of uncertainty and delay on the blockchain network and should be somehow avoided.

This is the reason why Algorand was created to solve most of these issues and accelerate the development of blockchain technology and its mass adoption. This is achieved by relying on algorithmic randomness for reaching a consensus between the nodes, which implies also its name - Algorand.[9] It combines PoS concepts with the Byzantine agreement protocol creating the unique consensus algorithm for Algorand called Pure Proof of Stake (PPoS).[9, 10] With it, the Algorand protocol achieves 125 times Bitcoin's throughput by staying at the same time secure and requiring insignificant resources to be operational.[13, 11] Instant transaction finality is also guaranteed because of the extremely low probability of the chain being forked. Thanks to this, transactions can be validated and accepted on the chain for less

[1] https://ethereum.org/en/developers/docs/smart-contracts/#limitations
[2] https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#creating-the-smart-contract
[3] https://developer.algorand.org/articles/contract-to-contract-calls-and-an-abi-come-to-algorand/

than a minute and the latency remains constant even when scaled up with more network participants.[13]

3.3.2 Pure Proof of Stake

As mentioned in [13], Algorand's security depends on the belief that the majority of the stake is in non-malicious hands. If more than 2/3 of the ALGO supply (Algorand's native currency) is owned by honest participants, then consensus can be achieved while tolerating malicious users and their attacks. This is possible because nodes are randomly and secretly selected to take part in the proposal and voting process. As long as an account has at least one ALGO in its possession it can be selected to participate in the consensus. The probability for this to happen is directly proportionate to the number of ALGO a node holds.

As can be observed in Figure 3.3, the process of reaching a consensus and appending a new block to the chain can be divided into 3 phases: selecting nodes for block proposals, filtering the proposals down to just one, and voting whether it should be added to the chain or rejected. Before the beginning of each phase, the nodes engage in something called cryptographic sortition to find out whether they are selected to participate or not. [10] The whole consensus process goes like this:

- Block proposal - For each ALGO that a node holds, a Verifiable Random Function[19] (VRF) is being computed. All nodes do this in parallel, secretly and independently. VRF is a cryptographic function that requires negligible time and resources and at the end outputs a hash value that will determine whether the node was chosen to propose a block or not. There will be also a short VRF proof that is easily verifiable from others that the proposal is truly valid and rightful.[10, 2] If the node was chosen, it combines some pending transactions into a new block and propagates it to the network using the gossip protocol (the protocol that nodes use for communication on the Algorand blockchain).[13, 10]
- Filtering down to a single block - Again all nodes run the VRF as part of the cryptographic sortition to establish a committee. For a specific time period, each of the selected nodes will receive at least a few proposals and its job is to check the VRF proof and compare which of all the blocks has the highest priority. In the end, the committee has to select only one block, which goes to the final phase.[10, 2]
- Block certification - A new committee is created again with the sortition algorithm. This time each ALGO that has a "winning" VRF hash will equate to one vote for this node. In other words, a node can vote a couple of times whether the new block should be added to the chain or not. The block is checked for things like overspending, double-spending, and any other flaws. If it is okay and the block gets a certain number of votes that are more than a certain threshold, then the block gets appended to the chain. This is the end of the round and in the next round, the whole process repeats itself. [13, 10, 2]

The reason why this consensus can operate securely even when there are some malicious nodes in the network is because of the cryptographic sortition. Each phase of the consensus process involves a different group of nodes, where each node is privately chosen to participate. In this way, an adversary doesn't know which node to corrupt or attack. By the time it is clear which nodes are part of the committee, it is already too late because their proposals or votes are already propagating through the network.[13, 2] It is important to note that all of these steps are happening within seconds, but even if the adversary is powerful and quick enough to execute attacks, it just won't know who to target. By virtue of its consensus mechanism, Algorand achieves speed, efficiency, security, and true decentralization.

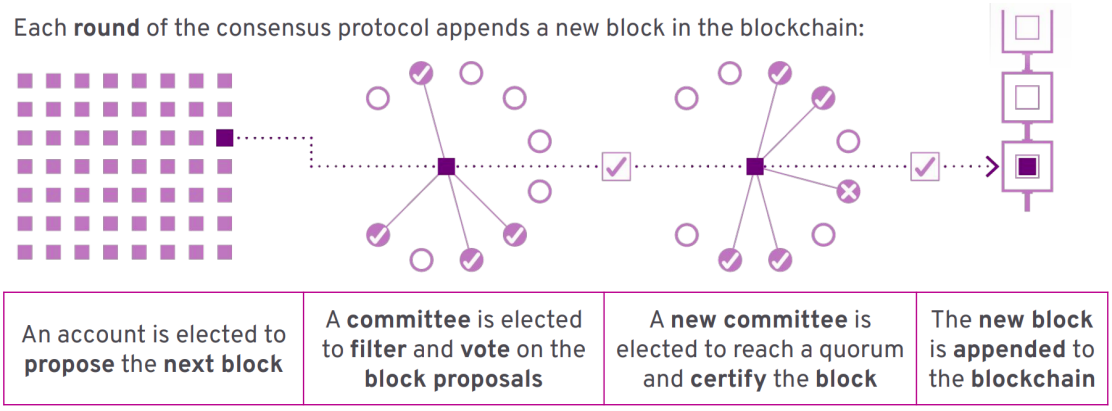

Figure 3.3: Pure Proof of Stake (Taken from [2])

3.3.3 Algorand smart contracts

Algorand is a programmable blockchain just like Ethereum. This means that smart contracts can also be written and evaluated on it. According to [7], in comparison to Ethereum, Algorand is more efficient and the execution of smart contracts happens for significantly less time and lower fees. There are two types of smart contracts for the Algorand blockchain - *stateful* and *stateless*.[6]

Stateful contracts live on the blockchain and hold some data (state) connected to it on the ledger. They can manipulate data by reading and writing it on the chain, execute and create transactions and perform logic operations as part of an application.[7, 6] Every such contract has an unique app ID and also an account that can hold ALGOs.

As mentioned in [27, 7] Stateless smart contracts (also known as Smart Signatures or Logic Signatures) are used for approving asset transactions between accounts. On Algorand, a transaction can be cryptographically signed either with the private key of the sender's account, a multi-signature (group of private keys of different accounts), or with a Logic Signature. These contracts aren't deployed to the blockchain as separate entities with their state, but rather attached to a transaction. The transaction will be successfully executed and accepted on the ledger only if the Logic Signature evaluates to true. Stateless smart contracts can be used also as escrow accounts between different parties by holding their funds and releasing them after some pre-agreed condition was met. Then the contract gets an address on the blockchain where the funds will be stored.[27, 7] In contrast to stateful contracts, they can't write/read state during evaluation.

The Algorand language used for writing and evaluating smart contracts and transactions on the blockchain is called Transaction Execution Approval Language (TEAL). TEAL is a low-level language with assembly-like syntax. Once the TEAL program is written and executed, it gets compiled into bytecode that is run on the Algorand Virtual Machine (AVM). The AVM is a Turing-complete execution environment on Algorand that works like a stack interpreter that processes TEAL programs and checks whether they are valid or not. If the program is valid, then the transaction will be sent to the network and the AVM will return a single non-zero value on top of the stack, signaling that the transaction was accepted.[7, 2]

Regarding the upgradability of the smart contracts[4], on Algorand they can be easily updated via a special type of transaction. If the transaction sender who wishes to update the contract is allowed to do so, then the new programs should be provided together with the app ID of the contract. However, this will update only the contract's logic, the state associ-

[4] `https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#update-smart-contract`

ated with it will remain immutable and can't be changed once the contract is deployed for the first time. This is a lot more convenient in comparison to other blockchains that require some kind of design pattern to achieve smart contract upgradability.

Contract to contract calling is also possible on the Algorand blockchain since the introduction of AVM 1.1 (TEAL version 6).[5] This can be done via inner transactions.[29] This allows developers to build complex applications on the Algorand network that can include several smart contracts collaborating. In the Implementation chapter of this thesis, all of these concepts like Stateful smart contracts, upgradability and contract-to-contract calling will be demonstrated in a real application.

3.4 Role-based programming

In today's society, we as humans tend to see ourselves and others playing distinct roles depending on different situations. Let's take for example a person named Bob. At work, Bob is an employee. At home, he is a husband and a father. When he is buying some groceries, then he is a customer of a store. These are all different roles that Bob plays according to the context, but all of the time this is the same person called Bob. The role-oriented approach tries to module this concept and view of our world into the process of developing software.

As noted in "Refactoring to Role Objects"[23] by F. Steinmann and F. Stolz, we can use the concept of Inheritance in OOP to model the example with the person Bob and his roles. We can make Bob the superclass and all of his roles of employee, husband, etc., could inherit from it as subclasses. Because each subclass can be extended with certain unique functionality needed for its context and at the same time inherits the basic behaviors and specifications of the person Bob, it could be argued that with Inheritance we successfully achieved modeling Bob and his roles as a program. However, as correctly pointed out in [23], now each new role that we create for Bob can be interpreted as an independent object representing a different individual which is not reasonable compared to the real-world scenario where every role is represented by the same person called Bob.

To make an adequate representation of the given scenario with Bob, we can use the Role Object pattern (ROP). In Figure 3.4 taken from [5], we have a UML diagram of ROP.With the explanations provided in [5] and our example of Bob and his roles, the structures in the diagram will have the following meaning:

- Component - this is an abstract class that shows how role objects will be handled by the ComponentCore and also some general methods applicable for all ComponentCore instances. In our case, we can name this construct *PersonAbstract*, which is a general representation of people in society with their roles and how to manage them.
- ComponentCore - this is where a concrete instance of a person can be created. Bob will be an instance of this class. We can rename it to *Person*.
- ComponentRole - All concrete roles derive from this abstract class. It makes the connection with the ComponentCore by storing a reference to it (for example a reference to the object Bob).
- ConcreteRole -here all concrete roles of Bob can be implemented with their additional functionalities as separate classes like Employee, Customer, Husband, Father, etc.

As described in [23] and [5], using ROP gives us the advantage to alter the behavior of an object dynamically depending on the context with the help of adding or removing role

[5] https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#contract-to-contract-calls

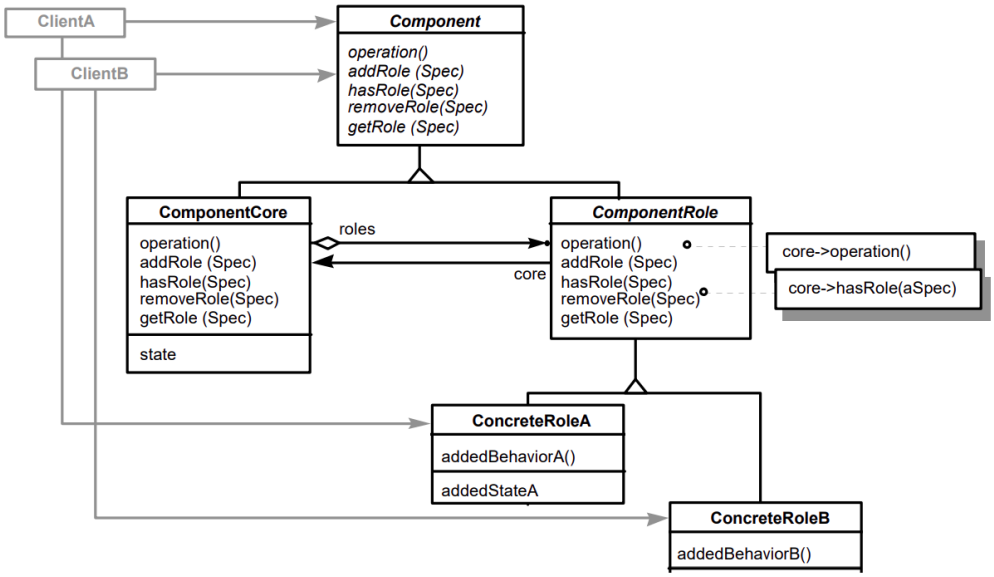

Figure 3.4: Structure diagram of the Role Object pattern (Figure 3 in [5])

objects at run time. In comparison to using Inheritance, the role objects will only extend the functionality of the core object and share its state, representing the same logical entity, namely the person Bob for instance. An object can hold many roles concurrently, each for its respective context. Each role can be also upgraded without the need to change anything in the core object or other roles. This means that with ROP we can achieve separation of contexts, low coupling, and easier code maintenance.

In summary, with role-based programming and Role Object pattern, we want to have objects that could have dynamically different attributes and behaviors according to specific contexts. This could be done when the object has different roles attached to it and can manage them at run time. Each role is independent and has its own purpose, but all of them collectively represent the same core object in different scenarios.

4 Implementation

In this chapter, the creation process of the PoC will be shown. First, we'll talk about Algorand's practical limits and what we need to consider before we start planning and developing the banking application. Then, we will present a detailed analysis and design of the application using UML diagrams. Finally, we will show key code snippets from the program that reflect important functionalities and the binding between the role-based application and the Algorand blockchain.

4.1 SDKs and TEAL limitations

To develop a meaningful role-based application, we will most likely need to use OOP concepts such as Inheritance, Composition, Abstract classes, etc. Unfortunately, the language TEAL with which we can create applications on the Algorand blockchain is a low-level language with an assembly-like syntax.[1] This means that TEAL doesn't provide a high level of abstraction and doesn't have these OOP concepts built in. Due to this reason, the process of creating complex applications using just TEAL can be very inconvenient, which may discourage many companies from using Algorand for their services. Fortunately, Algorand provides some software development kits (SDKs) that make the whole development process more pleasant and easy.[2] There are SDKs for some of the most popular programming languages like Python, Java, JavaScript, C#, Rust, etc. with which you can interact with the Algorand network.

For the PoC that I am going to create for this thesis, I will be using the Python SDK and PyTeal[3]. PyTeal is a domain-specific language for Algorand that allows us to write code for smart contracts in Python that is going to be compiled to TEAL afterward.[2, 7] Now we have a more convenient syntax for creating smart contracts and also a higher level of abstraction that we can utilize. It is important to reiterate that PyTeal programs are not compiled directly into bytecode and passed to the AVM for processing but are first compiled into TEAL programs.[2] This ultimately means that PyTeal is still dependent on the TEAL limitations. We can't compile classes and objects written in Python to TEAL, but at least now we have a binding mechanism between Algorand and a high-level programming language thanks to the SDK. This indicates the possibility for a role-based application to tightly interact and collaborate with the Algorand blockchain and its smart contracts.

[1] https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification/
[2] https://developer.algorand.org/docs/sdks/
[3] PyTeal Documentation (2022) https://pyteal.readthedocs.io/en/stable/index.html

4.2 Analysis and Design of the PoC

Thanks to the Python SDK, we can easily integrate applications that are written in Python with the Algorand blockchain. The bank application will be divided into on-chain and off-chain parts. In the blockchain world, on-chain means all of the functionality, values, and transactions that are going to be stored and executed on the blockchain itself. Off-chain will signify all the business logic that happens outside of the distributed ledger. As stated in the previous section, we can't implement OOP concepts on the Algorand blockchain directly. This means that the whole management of role objects will be handled via Python in an off-chain fashion. For this PoC we will have two separate roles that a person can play, a client and an investor. When a person plays the client role, he will have the possibility to open an account with a bank and fund it with ALGOs. Operations such as depositing and withdrawing are also available to the client. Transfers to other accounts are also realizable. In essence, a client can do all the standard operations that are available at a real-life banking institution. The investor, on the other hand, has the means and capabilities to create a bank and manage it. Each Person object that can have these two roles will be connected with an Algorand standalone account [4]. To send transactions, hold ALGOs, create smart contracts, etc. you need to have an account on the blockchain. This is one of the binding mechanisms between the objects of the Python application and the Algorand protocol. Each Person object will store its corresponding Algorand account as a list composed of two items - the Algorand account's public address and its private key. This information will be important later on for sending the transactions from the Python application to the blockchain via the SDK.

The roles can be dynamically added or removed. Moreover, the Person object can hold simultaneously both roles without a problem. To achieve this role-based approach, we are going to use the Role Object Pattern (ROP) and tailor it to our PoC scenario. In Figure 4.1, a UML class diagram shows what the section of the application that is responsible for the management of roles should look like. Thanks to this pattern, the Person object will now be able to add, remove and check its roles by inheriting from the PersonAbstract class. The object will store its current roles as key-value pairs in a Python dictionary.

As mentioned, each investor will have the opportunity to create a bank, but he can manage only one at a time for simplicity reasons of the PoC. As can be seen in the UML class diagram 4.2, each Bank object has its own corresponding smart contract. There are many methods defined in the Bank class, but their sole purpose is to convey the needed information for building a transaction that is going to be sent to the smart contract. However, to establish the connection between the off-chain and on-chain components, we need the Python SDK. All of the SDK functions that we require are located in the Transactions class which can be interpreted as the interface between the Python application and the Algorand blockchain.

All of the application's smart contracts are displayed on the right side of the class diagram 4.2. As pointed out in [18], smart contracts can be abstracted as OOP objects, having states and methods. Therefore, the contracts are represented as regular classes on the UML diagram. Smart contracts can read/write and store values on the blockchain. All values associated with the smart contract represent its state (represented as class attributes). There are global and local state storage [5]. In the global state, we can store important data that relates directly to the smart contract. This data can be used by the contract itself or by other contracts on the blockchain because it is globally readable. Values that relate to the participation of an account in a smart contract are stored in the local state. Every account stores this state on its account balance record, but only the smart contract can make changes to it.

[4] https://developer.algorand.org/docs/get-details/accounts/create/#standalone
[5] https://pyteal.readthedocs.io/en/stable/state.html

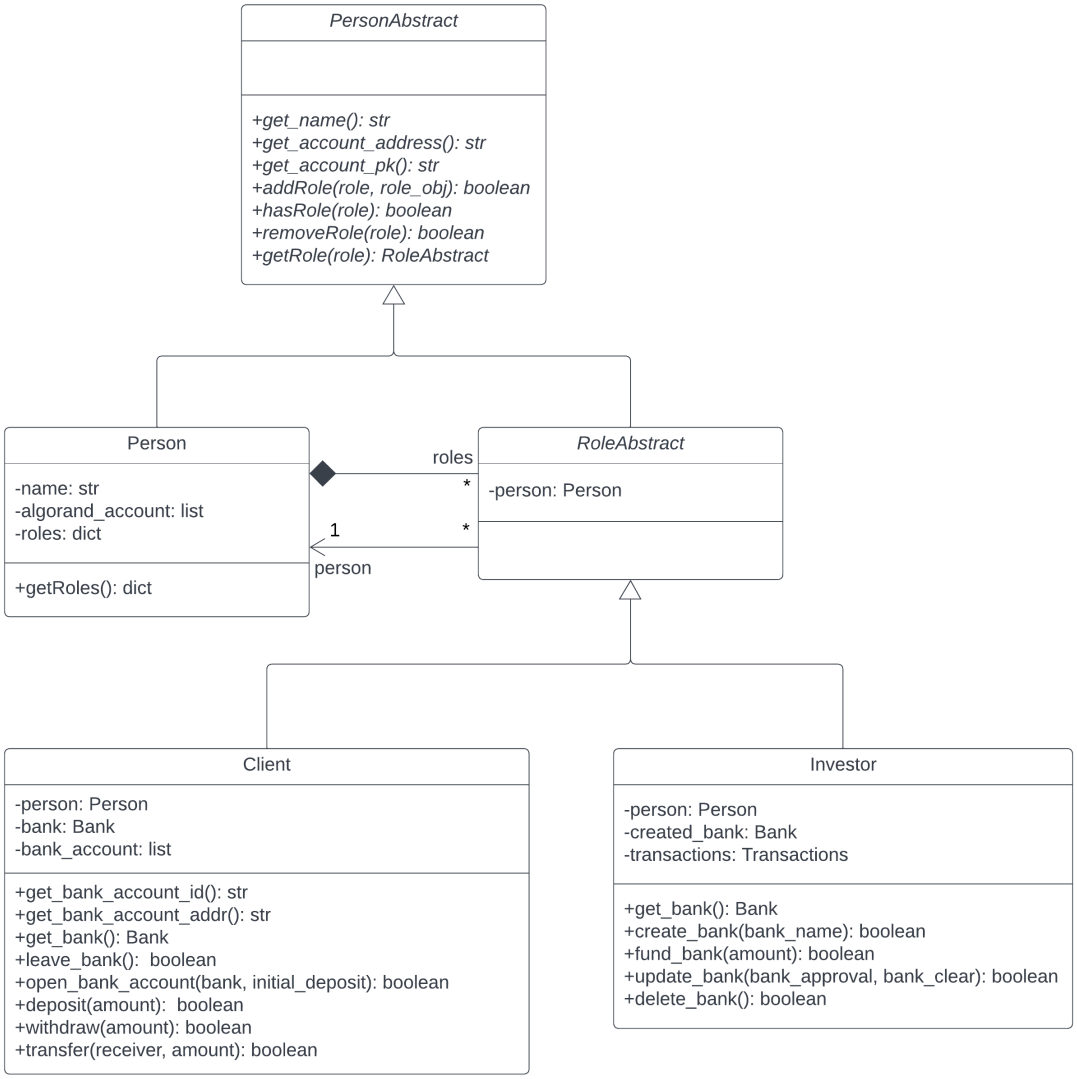

Figure 4.1: Role Object Pattern for the bank application

As can be seen in Figure 4.2, the bank smart contract is not the only one in this application and there are good reasons for that. It is always a good idea to follow the best practices of software development. I have tried to develop a modular and flexible PoC by spreading the functionality of the bank into separate smart contracts, each responsible for a different logical part. This makes the application easier to maintain, update and extend. On top of that, smart contracts on Algorand have a maximum size limit of 8KB which also has to be respected.[6] With this app structure, I will also be able to show the contract to contract calls on Algorand. Four additional contracts extend the functionality of the bank smart contract: Deposit, Withdraw, Transfer and Reference. The only one that needs more explanation is the Reference contract. Its sole purpose is to act as a template that is going to be replicated by every new client's bank account (also a smart contract). In other words, all bank accounts that are associated with the bank will have the same logic and state storage as defined in the Reference contract. The app ids and account addresses of these contracts are also stored in the Bank object and given as reference in a transaction when needed.

The way I have implemented the relationship between the bank smart contract and the other contracts is through the parent-child pattern.[28] The bank smart contract acts as the parent and it can create, update and delete additional contracts that act as its children. A counter stored in the global state of the parent called "children" will keep track of the contracts spawned by the bank. Almost all transactions sent by a client will first be processed in the bank and then redirected via contract to contract calling to a child that holds the necessary logic for further processing. The relationship between the bank and its children in the UML class diagram is represented through a Composition because the children contracts can't exist on their own or better said there are completely useless without the bank. Moreover, if an investor wants to close the bank and delete the bank smart contract, then all of its children should be destroyed first. If this is not the case, the transaction sent to delete the bank smart contract will fail. For a more detailed and graphical explanation of the communication between the bank and the secondary contracts, I have prepared UML sequence diagrams that should illustrate roughly the processes that take place in the application.

The sequence diagram 4.3 demonstrates what the process of opening a new bank account should look like. First, a method that belongs to the Client object is invoked and is provided with the amount of funding that the client will like to deposit in his new bank account. Then the client calls the Bank object giving his Algorand account credentials. The Bank object adds some additional information from itself and executes a function call to the Transactions object. There the data collected so far is packed into a group of two transactions - a payment and a call to the bank smart contract. They are finally sent to the blockchain with the help of the SDK methods. Now, the Algorand account connected to the Client object sends the payment and application call transaction to the bank smart contract. After checking some conditions, if everything was correct, the bank smart contract executes a transaction that creates a new smart contract representing the bank account of the client. A transaction object in JSON format is returned that is converted to a Python dictionary. The method of the Transactions object will return only the app id and account address of the newly created bank account. The Client object will store this information in a list. In this way, we create a link between an object of the Python application and a smart contract on Algorand. With this the operation of opening a bank account is successful.

The sequence diagram 4.4 shows the process of when the client wishes to deposit some funds into his existing bank account. Just like the procedure of opening a bank account, methods from the Client and Bank object are called and ultimately a transaction is sent to the blockchain via the SDK functions defined in the Transactions object. The bank smart contract receives a group transaction from the Algorand account connected to the Client

[6] https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#creating-the-smart-contract

object. The group is composed of a payment transaction (funds to be deposited) and a call to the bank smart contract. The bank does some checks on the validity of the transactions and when everything is in order, it calls the Deposit contract and sends the received ALGOs from the payment transaction. In the Deposit contract, some checks are being done again. It's usually just making sure that the one sending transactions to the contract is the bank that created it in the first place (the parent). If this is true, then the Deposit contract finally sends the deposited ALGOs to the corresponding bank account of the Client object. In the end, the Python application will receive a confirmation in the form of a boolean expression that signifies that the deposit was successful.

The withdraw functionality represented in 4.5 is in general the same thing that was described for the deposit operation. That's why I am going to rather explain directly the transfer functionality which is a little bit different (Sequence diagram 4.6). Until now, the transactions were always sent first to the bank smart contract and from there redirected to a child contract. Nevertheless, When a client wants to transfer some ALGOs from his bank account to another, then first the Transfer contract is called instead of the bank. This is done with the sole purpose to prevent re-entrancy which is not allowed on Algorand[7]. This means a contract can't call itself directly or indirectly using the contract to contract calling and this is done to hinder re-entrancy attacks. If the client wants to make a transfer from his bank account to another account that belongs to the same bank, then the bank smart contract will be visited two times, which results in re-entrancy and the transaction will be rejected. Therefore, we first call the Transfer contract. Next, with contract calls the operation will continue in the client's bank account and after that redirected to the bank where the receiver's account is. In the end, the bank of the receiver should transfer the funds to his bank account. The Client object should again get a boolean expression that will show whether the transaction was successful or not.

[7] `https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#contract-to-contract-calls`

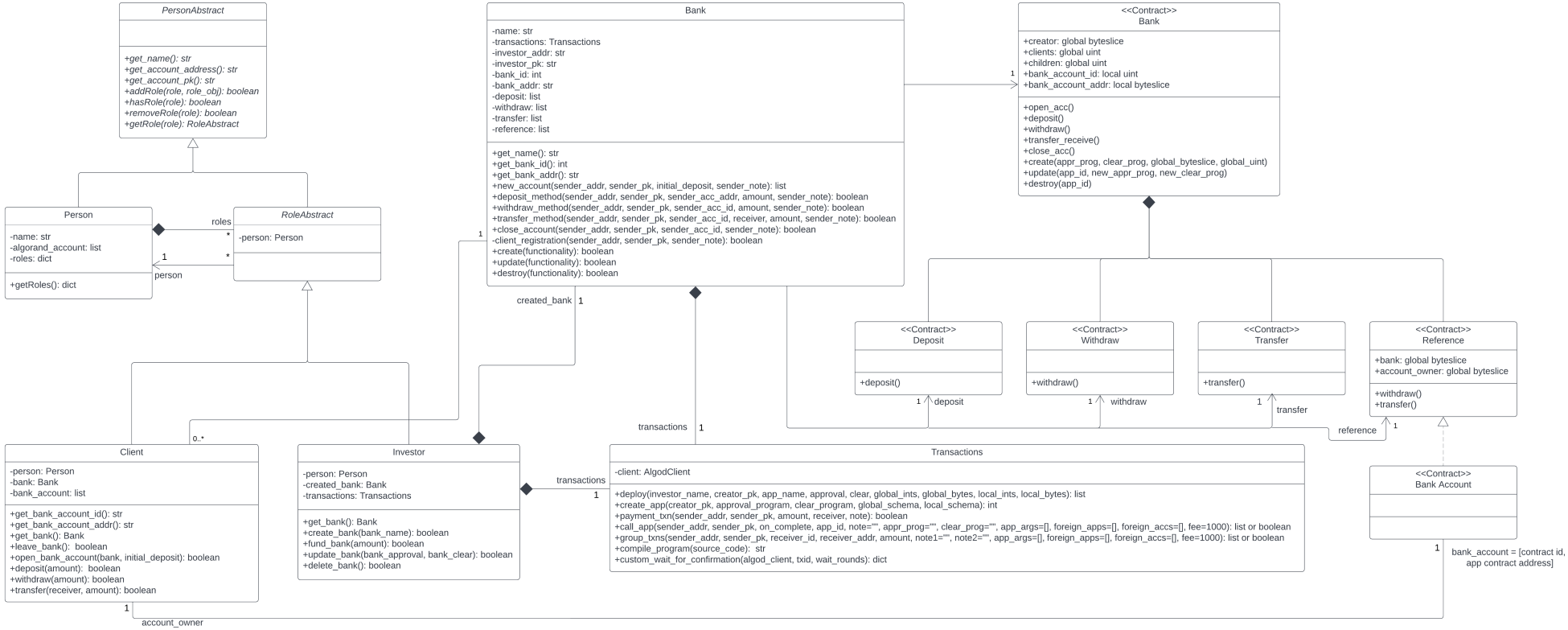

Figure 4.2: Detailed class diagram of the whole bank application

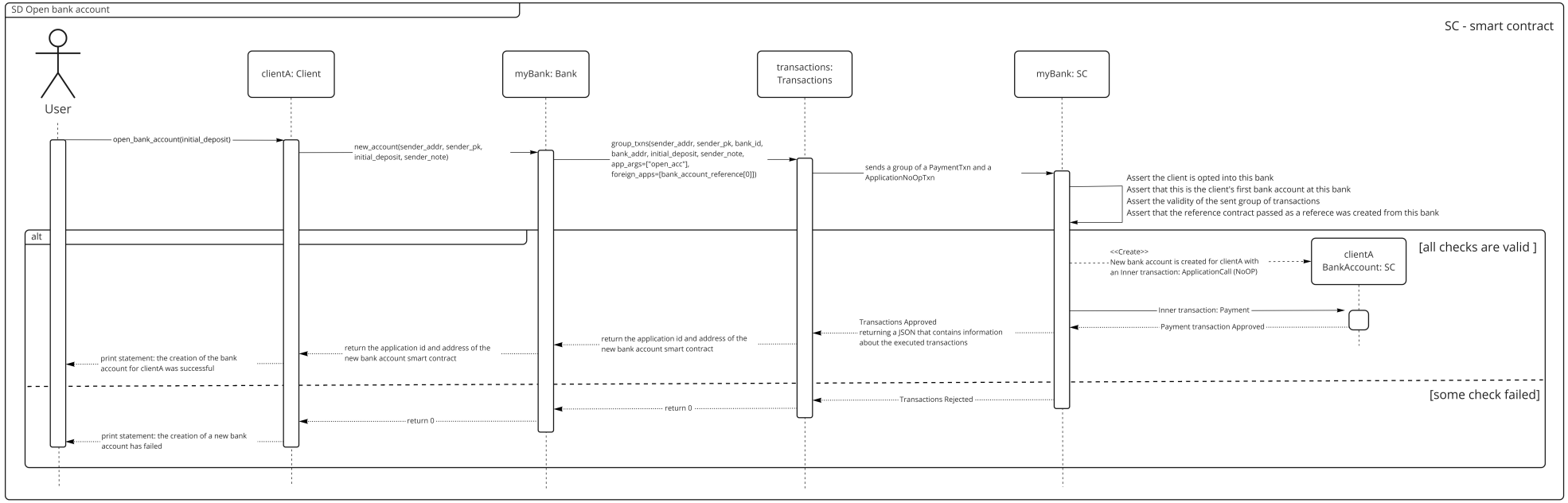

Figure 4.3: Sequence diagram showing the process of a client opening a bank account

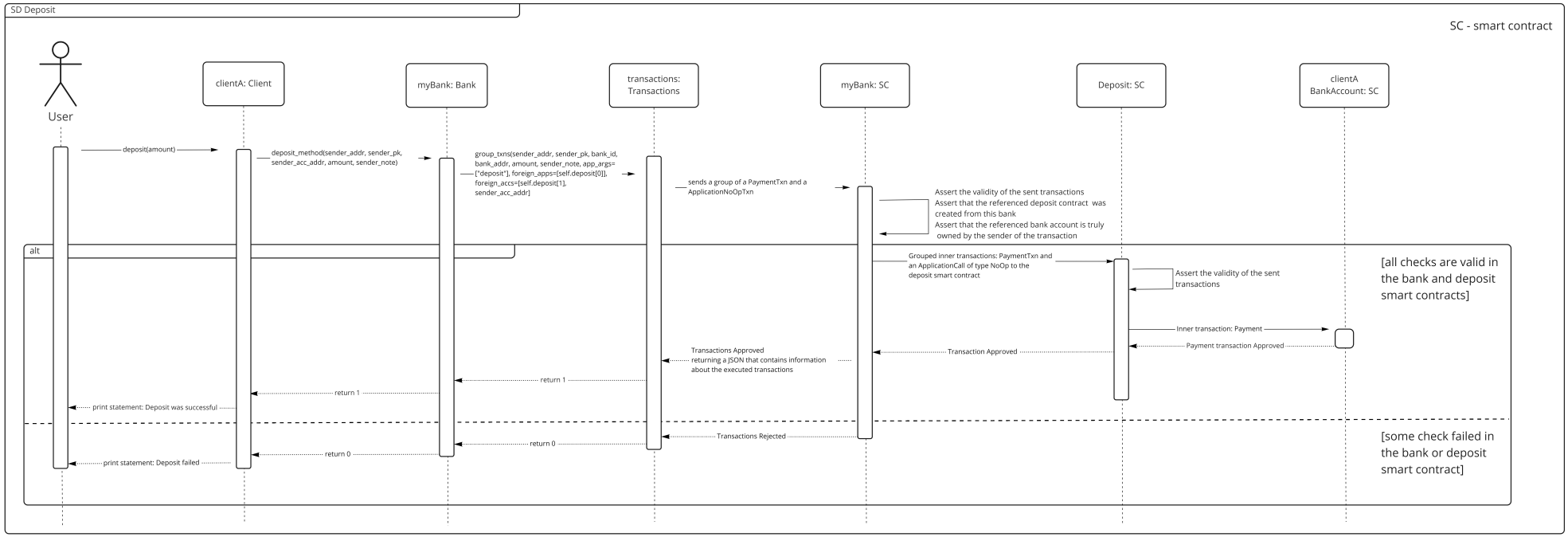

Figure 4.4: Sequence diagram showing the deposit functionality

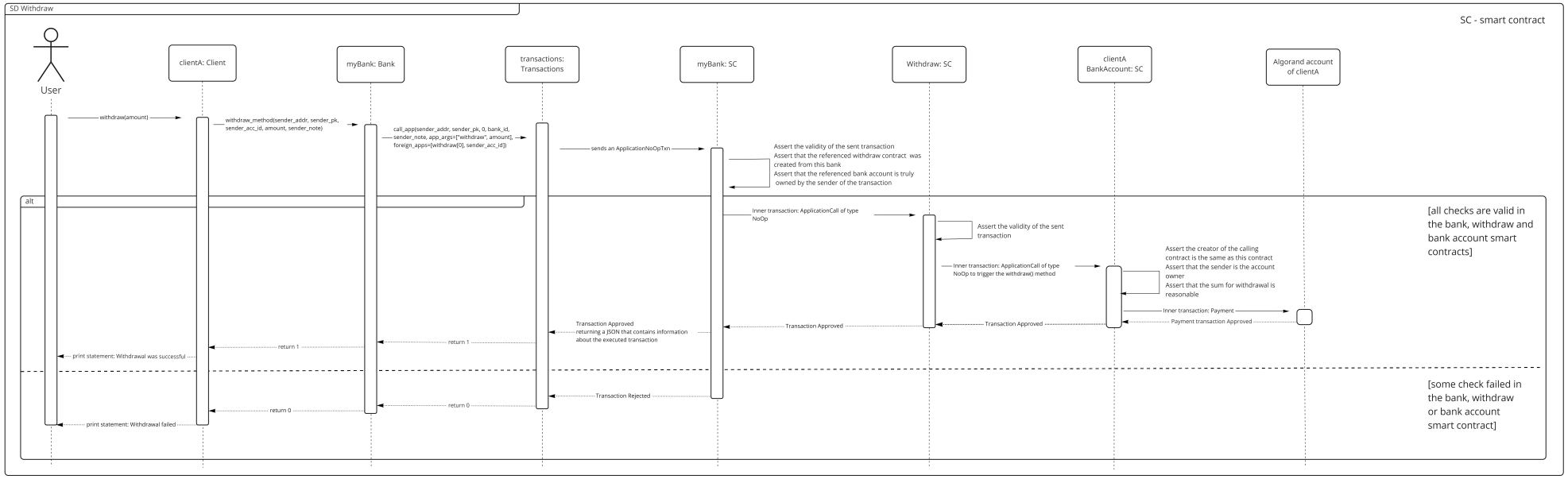

Figure 4.5: Sequence diagram showing the withdraw functionality

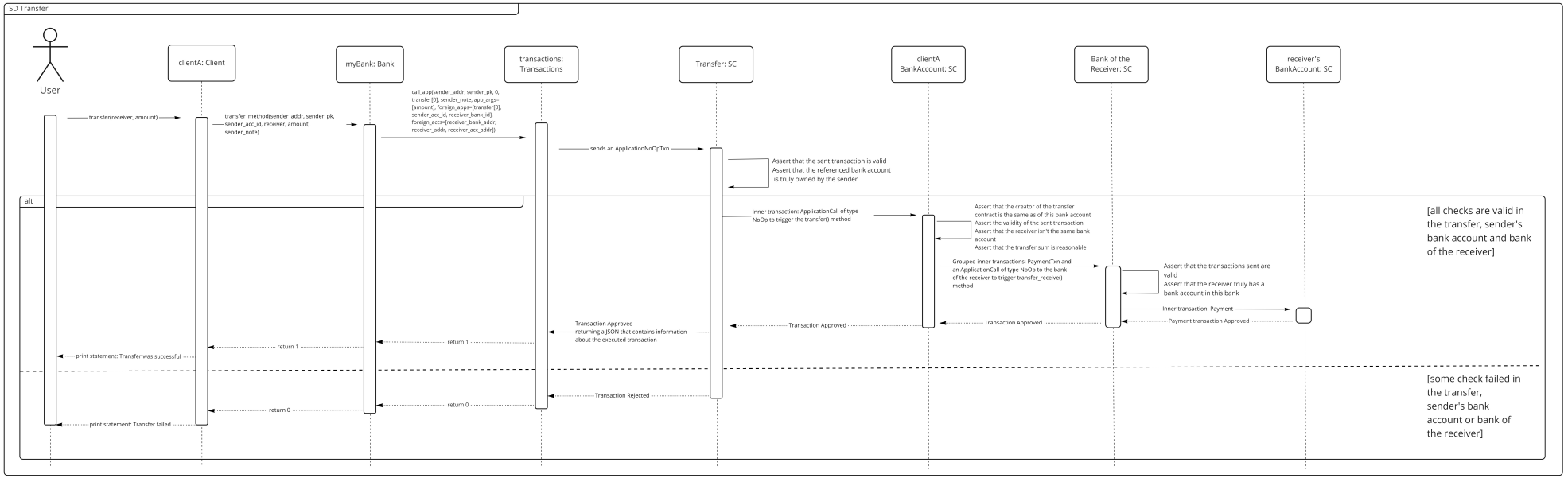

Figure 4.6: Sequence diagram showing the transfer functionality

4.3 Code implementation of the PoC

Having analyzed the general structure of the application using the UML diagrams, it is time to move on to the code implementation. As I mentioned before, I will use Python as a general-purpose language together with the Python SDK and PyTeal for writing smart contracts. Due to the large size and complexity of the application, I will focus only on certain parts that are important for answering the research questions of this thesis. Although some important technical features regarding Algorand will be addressed, a detailed line-by-line explanation of the application will not be presented. The main idea is simply to demonstrate that we can link a role-based application developed in Python together with Algorand and show how this is done. Anything else is unnecessary. Of course, a rough explanation of what does what will be given for each code snippet. Contract to contract calling and upgradability will also be presented. If you are further interested in understanding each line of the provided code, you can take a look at the official PyTeal[8] and Algorand documentation[9]. What will be shown is the process of creating and updating a bank together with the operation of opening a new bank account for a client. Additionally, the management of roles in the Python application via the Person object will also be presented. This will be sufficient to practically demonstrate all of the needed concepts. The code for the whole application can be found here[10].

4.3.1 Management of roles

A Person object should model after a real-life individual that can have different roles depending on the context. In the scenario of the PoC, the object can play either the role of a bank's client or as an investor that can create a bank. This role-based approach is implemented thanks to the ROP pattern shown in Figure 4.1, where the Person class will inherit from the PersonAbstract class how to manage role objects. The Python code for the Person class is the following:

```
class Person(PersonAbstract):

def __init__(self, name, algorand_account):
    self.name = name
    # holds [algorand account public addr, alogrand account private key]
    self.algorand_account = algorand_account
    # holds the different roles of this person as a dictionary
    self.roles = {}

...

def addRole(self, role, role_obj):
    if any(isinstance(x, type(role_obj)) for x in self.roles.values()):
        print("\n" + "{} already has the role {}".format(self.name, role))
        return False
    else:
        self.roles[role] = role_obj
        print("\n" + "\n" + "{} now has the role {}".format(self.name, role))
        return True

def hasRole(self, role):
    if role in self.roles.keys():
        print("\n" + "{} has the role {}".format(self.name, role))
        return True
```

[8] https://pyteal.readthedocs.io/en/stable/
[9] https://developer.algorand.org/docs/
[10] https://github.com/vasil241/bank_app/tree/master

```
25          else:
26              print("\n" + "{} role was not found".format(role))
27              return False
28
29      def removeRole(self, role):
30          if role in self.roles.keys():
31              del self.roles[role]
32              print("\n" + "The role {} was successfully removed".format(role))
33              return True
34          else:
35              print("\n" + "Removal failed! {} has no such role".format(self.name))
36              return False
37
38      def getRole(self, role):
39          if role in self.roles.keys():
40              return self.roles[role]
41          else:
42              print("\n" + "{} has no {} role".format(self.name, role))
43              return None
44
45      def getRoles(self):
46          if self.roles:
47              print("\n" + "{} currently has the roles: ".format(self.name) + ", ".join(self.
    roles.keys()))
48          else:
49              print("\n" + "{} has no roles".format(self.name))
50          return self.roles
```

Every Person object can add roles to itself by calling the *addRole(role, role_obj)* function. All of the role objects that extend the Person object will be stored in a dictionary using key-value pairs. That's why when we want to add a new role, we have to specify the key (role) and give the role object (role_obj) as a value. It is also important to note that for simplicity reasons of the PoC, a person can be a client of only one bank and an investor of only one bank. This means that a Person object can't have more than one client and investor role simultaneously. All of the other methods in this class are self-explanatory and define the management protocol of the roles. We can query, add, delete and get roles. Moreover, every Person object is connected to an Algorand account by holding information about its public address and private key in the list *algorand_account*.

We can easily create a new standalone Algorand account using the Python SDK and the code below. Then we can fund it with some test ALGOs from an Algorand dispenser[11]. This is necessary because, for the account to participate on the blockchain and execute transactions, it should hold at least 100.000 micro ALGOs (0.1 ALGO).[12]After we have generated an account, we just pass it into the constructor of the Person object during instantiation.

```
1 from algosdk import account
2
3 def generate_algorand_keypair():
4     private_key, address = account.generate_account()
5     print("Generated a new address: {}".format(address))
6     print("Generated a new private key: {}\n".format(private_key))
7     return [address, private_key]
```

[11] https://bank.testnet.algorand.network/
[12] https://developer.algorand.org/docs/get-details/accounts/create/

4.3.2 Bank creation

Now that we have defined the Person object and how it manages its roles, we can move on to the creation of the bank object and corresponding smart contract.

```
class Investor(RoleAbstract):
    def __init__(self, person):
        self.person = person
        self.created_bank = None
        self.transactions = Transactions()

    def create_bank(self, bank_name):
        if self.created_bank == None:
            investor_addr = self.get_account_address()
            investor_pk = self.get_account_pk()
            # transactions.deploy method should return id and addr of newly created bank [id, addr]
            l = self.transactions.deploy(self.get_name(), investor_pk, bank_name, bank_approval, bank_clear, 2, 1, 1, 1)
            self.created_bank = Bank(bank_name, investor_addr, investor_pk, l[0], l[1])
            if self.fund_bank(2000000):
                print("Investor successfully created and funded the bank with 2 Algos")
                return True
            else:
                print("Something went wrong with the creation or funding of the bank! Please remember that the investor needs at least 2 Algos for funding!")
                return False
        else:
            print("\nInvestor {} has already created a bank called {}".format(self.get_name() ,self.created_bank.get_name()))
            return False

    def fund_bank(self, amount):
        if self.created_bank == None:
            print("The investor hasn't created a bank yet! Funding not possible")
            return False
        if amount <= 0:
            print("Please add a valid amount for funding the bank!")
            return False

        investor_addr = self.get_account_address()
        investor_pk = self.get_account_pk()
        note = "Investor {} makes a {} micro algos payment transaction to {}".format(self.get_name(), amount, self.created_bank.get_name())
        print("\n" + note)
        val = self.transactions.payment_txn(investor_addr, investor_pk, amount, self.created_bank.get_bank_addr(), note)
        if val:
            print("Payment transaction was successful!")
            return True
        else:
            print("Payment transaction failed!")
            return False
```

After we add the investor role to the Person object, we can call the method *create_bank*. Inside it, we invoke the *deploy* method of the Transactions object. As parameters, we give the name of the new bank, the approval and clear programs that are necessary for the creation of a smart contract on Algorand (more on that later), and some integers at the end which represent how much global and local state storage should be allocated to this smart contract. If everything goes well, we should get as a return value from *deploy* a list

containing the app id and the account address of the newly created bank smart contract. Next, a Bank object is created that is going to be connected with the bank contract. The Investor then funds the newly created contract by calling *fund_bank(amount)*, which again uses the Transactions object to send a payment transaction to the bank smart contract with ALGOs from the account of the Person object. Now, let us look at how exactly we deploy this bank smart contract to the blockchain via the Transactions object and what exactly the *deploy* method looks like.

```
from algosdk.future.transaction import *
from algosdk.v2client import algod

class Transactions:
    def __init__(self):
        self.client = algod.AlgodClient("a" * 64, "http://localhost:4001")

    def deploy(self, investor_name, creator_pk, app_name, approval, clear, global_ints, global_bytes, local_ints, local_bytes):
        # declare bank application state storage (immutable)
        global_schema = StateSchema(global_ints, global_bytes)
        local_schema = StateSchema(local_ints, local_bytes)

        print("The investor {} is creating the bank {}...".format(investor_name, app_name))
        # Create bank
        app_id = self.create_app(creator_pk, approval, clear, global_schema, local_schema)
        app_addr = logic.get_application_address(app_id)
        #Display results
        print("Created the bank {} with id {} and address {}".format(app_name, app_id, app_addr))
        return [app_id, app_addr]

    def create_app(self, creator_pk, approval_program, clear_program, global_schema, local_schema):
        # get the address of the sender from its private key
        addr = account.address_from_private_key(creator_pk)
        # get node suggested parameters
        params = self.client.suggested_params()

        # compile approval teal to bytecode
        appr_bytes = self.compile_program(approval_program())
        # compile clear teal to bytecode
        clear_bytes = self.compile_program(clear_program())

        # Create the transaction
        # the 3rd argument represents OnComplete and 0 is for NoOp
        create_txn = ApplicationCreateTxn(addr, params, 0, appr_bytes, clear_bytes, global_schema, local_schema)
        # Sign it
        signed_txn = create_txn.sign(creator_pk)

        try:
            tx_id = self.client.send_transaction(signed_txn)
            # Wait for the result so we can return the app id
            result = self.custom_wait_for_confirmation(self.client, tx_id, 5)
            print("TXID: ", tx_id)
            return result['application-index']

        except Exception as err:
            print("Error message: {}".format(err))

        return 0
```

In the Transactions object, we define an *algod client*. Through it, the SDK allows the Python application to interact with the Algorand blockchain by making a connection with an Algorand node.[13] With the help of this node, we can send transactions and interact with the Algorand network. As shown in the code snippet of the Investor class, we're calling the *deploy* method here, which takes all of these parameters that I explained what they mean. With the SDK method *StateSchema* we limit the number of strings and integers that may be stored in the global and local state by using the integers passed as the last arguments in the *create_bank* method. After that, the *create_app* function is called that will deploy the smart contract. First, we build the transaction that is going to be sent to the blockchain via *ApplicationCreateTxn*. In there we pass some trivial parameters that the transaction needs to be valid. They can be obtained through the SDK function *suggested_params()*. Furthermore, we put in there the approval and the clear programs that are now compiled to bytecode through the *compile_program*. We also pass the global and local state schema and we specify what the type of transaction will be - in this case, a NoOp. With the *ApplicationCreateTxn* we now have a transaction that is ready to be sent to the blockchain. It is signed and authorized with the private key of the Person object's Algorand account and it is sent via the SDK method *send_transaction*. If everything goes as expected, we should get as a confirmation an app id that belongs to the new bank smart contract. Then, *create_app* will return this to the *deploy* method where with *get_application_address* we can derive also the account address of the smart contract. In the end, we return both the app id and account address to the Investor object and then things continue as explained earlier for the code snippet of the Investor class.

On Algorand there are 6 transaction types in total. For this PoC, we are interested only in the *Payment* and *Application Call* types. The Payment transaction is used to send ALGOs from one account to another. The Application Call is used to call smart contracts and trigger some logic in them. When sending such a transaction, an app id and an OnComplete method are given. The id is to specify which contract exactly we want to call and the OnComplete helps the contract decide which part of its logic should it execute. So, depending on the OnComplete, there are again six types of Application Call transactions[14]

- NoOp - a generic call to the smart contract.
- OptIn - a call that lets an account participate in a smart contract by enabling its local storage.
- DeleteApplication - deletes the application with the given app id.
- UpdateApplication - updates the approval and clear programs of the application.
- CloseOut - ends the participation of an account in a smart contract and deletes the local storage associated with it. However, it is possible that the contract doesn't allow it.
- ClearState - same as CloseOut, but it will clear the local state of the account associated with the smart contract no matter what.

It is important to note, that some of these types of transactions can be rejected from the smart contract, depending on how the developers have created it. For example, an application could reject all transactions that want to delete it or update it, this is perfectly possible.

[13] https://developer.algorand.org/docs/archive/build-apps/connect/
[14] https://developer.algorand.org/docs/get-details/transactions/#application-call-transaction

Now that we know which methods the investor uses to create a bank and how we deploy the bank smart contract via the Transactions object, it is time to take a look at the code of the smart contract itself.

```
1  def bank_approval():
2
3      ...
4
5      handle_setup = Seq(
6          App.globalPut(Bytes("creator"), Txn.sender()), # byteslice
7          App.globalPut(Bytes("clients"), Int(0)), # uint
8          App.globalPut(Bytes("children"), Int(0)), #uint
9          Approve()
10     )
11
12     ...
13
14     program = Cond(
15                     [Txn.application_id() == Int(0), handle_setup],
16                     [Txn.on_completion() == OnComplete.OptIn, Approve()],
17                     [Txn.on_completion() == OnComplete.CloseOut, on_closeout],
18                     [Txn.on_completion() == OnComplete.UpdateApplication, handle_update],
19                     [Txn.on_completion() == OnComplete.DeleteApplication, handle_delete],
20                     [Txn.on_completion() == OnComplete.NoOp, handle_noop]
21                 )
22
23     return compileTeal(program, Mode.Application, version=6)
24
25     def bank_clear():
26     program = Seq(
27         Assert(Txn.applications.length() == Int(1)),
28         close_acc(),
29         # decrement the number of clients that have a bank account with the bank
30         App.globalPut(Bytes("clients"), App.globalGet(Bytes("clients")) - Int(1)),
31         Approve()
32     )
33     return compileTeal(program, mode=Mode.Application, version=6)
```

The code that you see for the reference contract is written in PyTeal. As can be seen, it resembles Python syntax but also has its unique expressions. The two methods *bank_approval* and *bank_clear* are actually going to be compiled to two separate TEAL programs - approval and clear state (clear) program. Every Algorand stateful smart contract is composed of these two programs.[15] The approval program handles almost all type of Application Call transactions sent to the contract. Therefore, a big portion of the logic of the smart contract is written exactly there. The only type of transaction that the approval program doesn't handle is the ClearState. The clear state program is used solely for dealing with it. When a creation transaction of type NoOP is sent, the approval and clear programs will be used for the deployment of the smart contract and *handle_setup* will be immediately activated to assign values to the global state of the bank. Under the key *creator* the public address of the Algorand account that sent the creation transaction will be stored, the same account that belongs actually to the Person object that plays the investor role. When the investor wishes to delete the bank smart contract, all of the bank accounts of clients, and all of the additional contracts that were spawned from the bank should be first deleted. To keep track of both we have the counters *clients* and children (because of the parent-child pattern used). Between lines 14 and 21 is where the routing to different methods based on the type of the received trans-

[15] https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#the-lifecycle-of-a-smart-contract

action is happening. It can be handled with just a simple *Approve()* or *Reject()*, or with more complex logic. Line 23 is where we compile the PyTeal code into an approval TEAL program. The textitbank_clear is a lot simpler. It will be used when a client wishes to abruptly close his bank account and clear his local state connected to the smart contract. Of course, first, we return all of the funds that remain on the bank account smart contract to the client's Algorand account and delete it. Then we can reduce the counter *clients* in the global state by 1 and then approve the wish of the client to leave the bank.

Now that we have cleared the basic structure of a smart contract, the different types of transactions, and how the bank is being created, we will take a look at how the bank can spawn additional contracts for its use. These contracts will hold logic that will help the bank to forward its operations outside itself. We will examine the deployment process of the Reference contract because the bank will need it for the creation of bank accounts for clients. It all starts from the Bank object's method *create* where we specify which child contract we want to create - Deposit, Withdraw, Transfer or Reference. Then via the *call_app* function of the Transactions object, we will send a NoOp transaction to the bank smart contract. The transaction will be routed by the approval program of the bank to the *create* function.

```
1  def create(appr_prog, clear_prog, global_byteslice, global_uint):
2      return Seq(
3          # only the admin that created the bank can execute the logic inside this method
4          Assert(Txn.sender() == Global.creator_address()),
5          Assert(Len(appr_prog)),
6          Assert(Len(clear_prog)),
7          InnerTxnBuilder.Begin(),
8          InnerTxnBuilder.SetFields({
9              TxnField.type_enum: TxnType.ApplicationCall,
10             TxnField.on_completion: OnComplete.NoOp,
11             TxnField.approval_program: appr_prog,
12             TxnField.clear_state_program: clear_prog,
13             TxnField.global_num_byte_slices: Btoi(global_byteslice),
14             TxnField.global_num_uints: Btoi(global_uint),
15             # this field is only because of the reference contract requirement in setup
16             TxnField.accounts: [Txn.sender()],
17             TxnField.note: Bytes("Bank deploys a new child/functionality (additional smart
       contract)"),
18         }),
19         InnerTxnBuilder.Submit(),
20         child_addr := AppParam.address(InnerTxn.created_application_id()),
21         Assert(child_addr.hasValue()),
22         # fund the newly created child smart contract with funds from the bank
23         InnerTxnBuilder.Begin(),
24         InnerTxnBuilder.SetFields({
25             TxnField.type_enum: TxnType.Payment,
26             TxnField.amount: Int(100000), # 0.1 Algo is more than enough
27             TxnField.receiver: child_addr.value(),
28             TxnField.note: Bytes("Fund the newly created child smart contract / additional
       functionality"),
29         }),
30         InnerTxnBuilder.Submit(),
31         App.globalPut(Bytes("children"), App.globalGet(Bytes("children")) + Int(1)),
32         Approve()
33     )
```

In this method, we are passing the approval and clear programs of the child contract together with its global state schema. It is important to note, that only the investor can execute this function and allow the creation of additional contracts for the bank. This is done by doing a check on line 4 if the account sending the transaction was indeed the creator of this

bank smart contract. The next important step is the inner transaction that is being built and will deploy the Reference contract. Precisely with the help of inner transactions contract to contract calling is being achieved[16]. They allow a smart contract to deploy another contract, send an Application Call or Payment transaction, and in general execute all types of transactions from itself automatically. With this powerful concept, we can build complex and modular applications on the Algorand blockchain without a problem. In this case, the bank smart contract will be sending 2 consecutive transactions, one for the deployment of the Reference contract and the other for fundings its account with 0.1 ALGO to cover its minimum balance. In the end, we will increase the counter *children* with one, because the bank smart contract now has 1 additional contract that it is using for its operations. Now it will be also a good moment to explain what the *note* field stands for. Notes are used for storing additional information about the transaction. They can be viewed as a form of metadata on the blockchain. In this PoC I am mainly using it to write messages that describe what the transaction is doing and what is its goal. In the end, these notes can be found and read when examining blocks on the ledger.

Now that we have a bank and a Reference contract that will be used as a template for each new bank account, we can move on to describe the process of opening a bank account. But before that, let us examine how easy it is to update an existing smart contract on the Algorand blockchain without the need for any design patterns.

```
def update_bank(self, bank_approval, bank_clear):
        if self.created_bank == None:
            print("The investor hasn't created a bank yet! Update not possible")
            return False
        investor_addr = self.get_account_address()
        investor_pk = self.get_account_pk()
        appr_bytes = self.transactions.compile_program(bank_approval())
        clear_bytes = self.transactions.compile_program(bank_clear())
        bank_id = self.created_bank.get_bank_id()
        note = "The investor {} updates the bank {}".format(self.get_name(), self.
    created_bank.get_name())
        print("\n" + note)
        val = self.transactions.call_app(investor_addr, investor_pk, 4, bank_id, note,
    appr_bytes, clear_bytes)
        if val:
            print("Bank was successfully updated")
            return True
        else:
            print("Bank update operation failed!")
            return False
```

The update operation for a bank smart contract starts with the investor. Again, we are passing on some important information to the Transactions object where the transaction will be built and sent to the bank smart contract. The key part here is providing a new approval and clear state program with which the bank will be updated. After the Application Call transaction of type UpdateApplication is received by the bank's current approval program, it will be re-routed to the *handle_update*. As can be seen in the code below, there is only a check to make sure that the bank is going to be updated, because of the wish of its creator, the investor. No one else has permission to update the bank and the transaction in such cases will be immediately rejected.

```
handle_update = Seq(
```

[16] https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#inner-transactions

```
        # make sure the update call is coming from the admin who created the bank
        Assert(Txn.sender() == Global.creator_address()),
        Approve()
    )
```

After the check is valid, there is just a simple *Approve* for giving positive confirmation of the transaction which will change the old approval and clear programs of the bank with the new ones, swapping the logic of the bank smart contract. This is the elegant way in which Algorand provides contract upgradability. It is important to understand that the contract remains the same, the app id and its account address aren't changed. The state storage also remains immutable and can't be influenced by this update. Thanks to this feature, developers can write and deploy smart contracts on Algorand without much concern and with the reassurance that they can easily optimize and update certain parts later on without much effort or using complex design patterns.

4.3.3 The process of opening a bank account

Every client has the option to open a bank account with an existing bank. For this purpose, the bank needs also to have a Reference contract as a child. It will act as a general template that all smart contracts that represent bank accounts will replicate when deployed. The operation of opening a new bank account starts with the Client object calling the *open_bank_account* method.

```
def open_bank_account(self, bank, initial_deposit):
        sender_addr = self.get_account_address()
        sender_pk = self.get_account_pk()
        if initial_deposit < 100000:
                print("Initial deposit for creating bank account should be at least 0.1 Algo!")
                return False
        if self.bank == None:
            sender_note = "{} wants to be a client of bank {}".format(self.get_name(), bank.get_name())
            print("\n" + sender_note)
            reg = bank.client_registration(sender_addr, sender_pk, sender_note)
            if reg:
                self.bank = bank
                print("Registration as a client in the bank was successful")
            else:
                print("Registration as a client in the bank failed!")
                return False
            sender_note = "{} wishes to open a bank account with {}".format(self.get_name(), self.bank.get_name())
            print("\n" + sender_note)
            l = self.bank.new_account(sender_addr, sender_pk, initial_deposit, sender_note)
            if l:
                print("A new bank account with id {} and address {} was created for the client {}".format(l[0], l[1], self.get_name()))
                self.bank_account = l
                return True
            else:
                print("The operation of opening a new bank account with {} has failed!".format(self.bank.get_name()))
                self.bank = None
                return False
        else:
            print("Client already has a bank account in the bank {}!".format(bank.get_name()))
```

```
30            return False
```

Each client willing to open a bank account will need to first opt into the corresponding bank smart contract. That's why we execute the function *client_registration* of the Bank object that again via the SDK methods in the Transactions object will send a transaction of type OptIn to the bank smart contract. With this, the client's Algorand account will now have local storage associated with the bank contract where data regarding its bank account can be stored. After this is successfully done, the client can call the Bank object's *new_account* method and pass the needed information as parameters for the next step.

```
1  def new_account(self, sender_addr, sender_pk, initial_deposit, sender_note):
2          # should return the id and addr of the newly created bank account
3          if self.reference == None:
4              print("Reference contract is still not created, can't create a bank account!")
5              return False
6          l = self.transactions.group_txns(
7              sender_addr, sender_pk, self.bank_id, self.bank_addr, initial_deposit,
       sender_note, app_args=["open_acc"], foreign_apps=[self.reference[0]]
8              )
9          return l
```

Here is a good opportunity to explain what are *applications, accounts* and *arguments* array.[17] When the smart contract that we are calling is going to interact with other contracts further down the line and read their global state, their app ids should be given as reference in the applications array. The same goes for accounts in the accounts array, when we need to read their local storage or balance information. The reason behind this is that the AVM needs to pre-load all the data associated with these applications or accounts into memory before evaluation. This is done purely to achieve better efficiency on the Algorand blockchain. The maximum number of foreign accounts that can be referenced per transaction is 4 and for applications, it is 8. In the arguments array, we pass all other additional data that can be needed and used from the smart contract. For example in the arguments array when calling the /textittransactions.group_txns the string 'open_acc' is passed. This will help the bank smart contract navigate itself which method exactly to call in its approval program. Moreover, the app id of the Reference contract is also given in the applications array, because the bank smart contract will need it during the deployment of a new bank account. So, the Bank object will further call the *group_txns* method from the Transactions object to send a group of transactions to the bank smart contract.

```
1  def group_txns(self, sender_addr, sender_pk, receiver_id, receiver_addr, amount, note1="",
       note2="", app_args=[], foreign_apps=[], foreign_accs=[], fee=1000):
2          atc = AtomicTransactionComposer()
3          signer = AccountTransactionSigner(sender_pk)
4          params = self.client.suggested_params()
5          params.fee = fee
6
7          pay_txn = PaymentTxn(sender_addr, params, receiver_addr, amount, note=note1)
8          txn1 = TransactionWithSigner(pay_txn, signer)
9          atc.add_transaction(txn1)
10
11         call_txn = ApplicationNoOpTxn(sender_addr, params, receiver_id, app_args,
       foreign_accs, foreign_apps, note=note2)
12         txn2 = TransactionWithSigner(call_txn, signer)
```

[17] https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/#smart-contract-arrays

```
        atc.add_transaction(txn2)

        try:
            result = atc.execute(self.client, 5)
            txn1_result = self.client.pending_transaction_info(result.tx_ids[0])
            txn2_result = self.client.pending_transaction_info(result.tx_ids[-1])
            print("TXID 1: ", result.tx_ids[0])
            print("TXID 2: " ,result.tx_ids[-1])

            if 'inner-txns' in txn2_result:
                if 'application-index' in txn2_result['inner-txns'][0]:
                    id = txn2_result['inner-txns'][0]['application-index']
                    addr = logic.get_application_address(id)
                    return [id, addr]
                else:
                    return 1
            else:
                return 1

        except Exception as e:
            print("Error message: {}".format(e))
            pass

        return 0
```

An important feature of Algorand is the possibility of grouping transactions as an Atomic transfer.[18] This means that transactions that are part of the group will all succeed or all of them will fail. It is important to note, that grouping transactions won't decrease the cost or time of execution, but will guarantee their mutual confirmation or rejection which can be useful for some use cases. Certain steps need to be followed for the creation of an Atomic transfer, but using the Atomic Transaction Composer[19] can simplify the process. It is again a function of the SKD that makes things more convenient. We will group a Payment and an Application call of type NoOp transactions. The payment transaction is for transferring ALGOs from the client's Algorand account to the bank smart contract as an initial deposit for the new bank account. What we want is that when a client opens a bank account, he must also necessarily put some money in it, just like it's done with real-life banks. In the end, if everything went well, the *group_txns* should return the app id and account address of the new bank account created on the blockchain. Let's have a look now at how the bank smart contract will process this group of transactions.

```
def open_acc():
    # a correct call to open_acc() needs to include also a payment transaction in addition
    to the app call
    deposit, app_call = Gtxn[0], Gtxn[1]
    gtxn_check = Seq(
        Assert(Global.group_size() == Int(2)),
        Assert(deposit.type_enum() == TxnType.Payment),
        Assert(deposit.receiver() == Global.current_application_address()),
        Assert(deposit.close_remainder_to() == Global.zero_address()),
        Assert(app_call.type_enum() == TxnType.ApplicationCall),
        Assert(app_call.on_completion() == OnComplete.NoOp),
        Assert(app_call.application_id() == Global.current_application_id()),
        # applications[1] represent the logic in reference.py
        # this is the smart contract that each bank account replicates
        Assert(app_call.applications.length() == Int(1))
    )
```

[18] https://developer.algorand.org/docs/get-details/atomic_transfers/
[19] https://developer.algorand.org/docs/get-details/atc/

```
16
17      return Seq(
18          # client has to first be opted into the bank
19          Assert(App.optedIn(Txn.sender(), Global.current_application_id())),
20          # acc_check is used to check if the sender has already opened a bank account, more
    than 1 is not allowed
21          If(App.localGet(Txn.sender(), Bytes("bank_account_id")), Reject()),
22          gtxn_check,
23          # the reference contract should be a child of the bank (parent)
24          creator := AppParam.creator(app_call.applications[1]),
25          Assert(Global.current_application_address() == creator.value()),
26          # we take the programs and state schema of the reference contract to create a new
    bank account
27          approval_prog := AppParam.approvalProgram(app_call.applications[1]),
28          clear_prog := AppParam.clearStateProgram(app_call.applications[1]),
29          global_byteslices_number := AppParam.globalNumByteSlice(app_call.applications[1]),
30          Assert(approval_prog.hasValue()),
31          Assert(clear_prog.hasValue()),
32          Assert(global_byteslices_number.hasValue()),
33          InnerTxnBuilder.Begin(),
34          InnerTxnBuilder.SetFields({
35              TxnField.type_enum: TxnType.ApplicationCall,
36              TxnField.on_completion: OnComplete.NoOp,
37              TxnField.approval_program: approval_prog.value(),
38              TxnField.clear_state_program: clear_prog.value(),
39              TxnField.global_num_byte_slices: global_byteslices_number.value(),
40              TxnField.global_num_uints: Int(0), # no global uints in bank account
41              # client addr in foreign accounts to create a connection between client and
    bank account
42              TxnField.accounts: [Txn.sender()],
43              TxnField.note: Bytes("Creates a new bank account for the client"),
44              TxnField.fee: Int(0)
45              }),
46          InnerTxnBuilder.Submit(),
47          # get the app id and address of the newly created bank account
48          App.localPut(Txn.sender(), Bytes("bank_account_id"), InnerTxn.
    created_application_id()),
49          acc_addr := AppParam.address(InnerTxn.created_application_id()),
50          Assert(acc_addr.hasValue()),
51          App.localPut(Txn.sender(), Bytes("bank_account_addr"), acc_addr.value()),
52          # fund the bank account with the client's initial deposit
53          InnerTxnBuilder.Begin(),
54          InnerTxnBuilder.SetFields({
55              TxnField.type_enum: TxnType.Payment,
56              TxnField.amount: deposit.amount(),
57              TxnField.receiver: acc_addr.value(),
58              TxnField.note: Bytes("Funds the newly created bank account"),
59              TxnField.fee: Int(0),
60          }),
61          InnerTxnBuilder.Submit(),
62          # increase the number of the bank's clients by 1
63          App.globalPut(Bytes("clients"), App.globalGet(Bytes("clients")) + Int(1)),
64          Approve()
65      )
```

The first thing that is done in the *open_acc* method of the bank smart contract is to do some checks on the transactions sent. This is to make sure that all the conditions that are needed for opening a bank account are met. The important part here is the inner transactions submitted by the bank. In the first one, the approval program and clear state program of the Reference contract are taken as a template together with its state storage schema. The bank account that is going to be deployed with this inner transaction is going to be an

exact copy of the Reference contract. After the bank account is created successfully, we are storing its app id as a value in the local state of the client's Algorand account under the key *bank_account_id*. Then we derive also the address of the newly created contract and again save it under *bank_account_addr* in the local state. This way the bank maps the created bank account together with the standalone account of the bank's client. The second inner transaction acts as a payment transaction to the newly created bank account by transferring the funds that the client wished to deposit initially. After all of this, the client now has a funded bank account and the operation was successful. The bank increases the counter *clients* in its global state by 1 and this was the whole process of opening a bank account.

5 Results and discussion

6 Conclusion

Bibliography

[1] Adnan, Imeri ; Lamont, Jonathan ; Agoulmine, Nazim ; Khadraoui, Djamel: Model of dynamic smart contract for permissioned blockchains. In: *2019 Practice of Enterprise Modelling Conference Forum (PoEM 2019 Forum)* Bd. 2586, 2019

[2] Bassi, Cosimo: *Algorand School*. 2022. – `https://github.com/cusma/algorand-school/blob/main/algorand-school-english.pdf`

[3] Bui, Van C. ; Wen, Sheng ; Yu, Jiangshan ; Xia, Xin ; Haghighi, Mohammad S. ; Xiang, Yang: Evaluating Upgradable Smart Contract. In: *2021 IEEE International Conference on Blockchain (Blockchain)*, 2021, S. 252–256

[4] Buterin, Vitalik: *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. 2014. – `https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf`

[5] Bäumer, Dirk ; Riehle, Dirk ; Siberski, Wolf ; Wulf, Martina: The Role Object Pattern. In: *Washington University Dept. of Computer Science*, 1998

[6] Chaudhury, Archie ; Haney, Brian: Algorand Autonomous. In: *SSRN* (2021). – https://ssrn.com/abstract=3819055

[7] Chaudhury, Archie ; Haney, Brian: Smart Contracts on Algorand. In: *SSRN* (2021). – https://ssrn.com/abstract=3887719

[8] Chaum, David: Computer Systems Established, Maintained and Trusted by Mutually Suspicious Groups, Ph.D. dissertation, Dept. Comput. Sci., Univ. California, Berkeley, 1982

[9] Chen, Jing ; Micali, Silvio: Algorand: A secure and efficient distributed ledger. In: *Theoretical Computer Science* 777 (2019), S. 155–183. – ISSN 0304–3975

[10] Conti, Mauro ; Gangwal, Ankit ; Todero, Michele: Blockchain Trilemma Solver Algorand Has Dilemma over Undecidable Messages. In: *Proceedings of the 14th International Conference on Availability, Reliability and Security*, Association for Computing Machinery, 2019 (ARES '19). – ISBN 9781450371643

[11] Fooladgar, Mehdi ; Manshaei, Mohammad H. ; Jadliwala, Murtuza ; Rahman, Mohammad A.: On Incentive Compatible Role-Based Reward Distribution in Algorand. In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, S. 452–463

[12] Gervais, Arthur ; Karame, Ghassan O. ; Wüst, Karl ; Glykantzis, Vasileios ; Ritzdorf, Hubert ; Capkun, Srdjan: On the Security and Performance of Proof of Work Blockchains. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA : Association for Computing Machinery, 2016 (CCS '16). – ISBN 9781450341394, S. 3–16

[13] Gilad, Yossi ; Hemo, Rotem ; Micali, Silvio ; Vlachos, Georgios ; Zeldovich, Nickolai: Algorand: Scaling byzantine agreements for cryptocurrencies. In: *Proceedings of the 26th symposium on operating systems principles*, 2017, S. 51–68

[14] Haber, Stuart ; Stornetta, Wakefield: How to time-stamp a digital document. In: *Journal of Cryptology* 3 (1991), S. 99–111

[15] Hassani, Hossein ; Huang, Xu ; Silva, Emmanuel: Banking with blockchain-ed big data. In: *Journal of Management Analytics* 5 (2018), Nr. 4, S. 256–275

[16] Khan, Abdul G. ; Zahid, Amjad H. ; Hussain, Muzammil ; Farooq, M ; Riaz, Usama ; Alam, Talha M.: A journey of WEB and Blockchain towards the Industry 4.0: An Overview. In: *2019 International Conference on Innovative Computing (ICIC)*, 2019, S. 1–7

[17] Kshetri, Nir: Potential roles of blockchain in fighting poverty and reducing financial exclusion in the global south. In: *Journal of Global Information Technology Management* 20 (2017), Nr. 4, S. 201–204

[18] Mattis, Toni ; Hirschfeld, Robert: Activity Contexts: Improving Modularity in Blockchain-Based Smart Contracts Using Context-Oriented Programming, Association for Computing Machinery, 2018 (COP '18). – ISBN 9781450357227, S. 31–38

[19] Micali, S. ; Rabin, M. ; Vadhan, S.: Verifiable random functions. In: *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, 1999, S. 120–130

[20] Nakamoto, Satoshi: Bitcoin: A Peer-to-Peer Electronic Cash System. (2008)

[21] Panwar, Arvind ; Bhatnagar, Vishal: Distributed Ledger Technology (DLT): The Beginning of a Technological Revolution for Blockchain. In: *2nd International Conference on Data, Engineering and Applications (IDEA)*, 2020, S. 1–5

[22] Saleh, Fahad: Blockchain without Waste: Proof-of-Stake. In: *The Review of Financial Studies* 34 (2020), Nr. 3, S. 1156–1190. – ISSN 0893–9454

[23] Steimann, Friedrich ; Stolz, Fabian U.: Refactoring to Role Objects. In: *Proceedings of the 33rd International Conference on Software Engineering*, Association for Computing Machinery, 2011. – ISBN 9781450304450, S. 441–450

[24] Szabo, Nick: Formalizing and Securing Relationships on Public Networks. In: *First Monday* 2 (1997), Nr. 9

[25] Vokerla, Rahul R. ; Shanmugam, Bharanidharan ; Azam, Sami ; Karim, Asif ; Boer, Friso D. ; Jonkman, Mirjam ; Faisal, Fahad: An Overview of Blockchain Applications and Attacks. In: *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, 2019, S. 1–6

[26] Wang, Shuai ; Ouyang, Liwei ; Yuan, Yong ; Ni, Xiaochun ; Han, Xuan ; Wang, Fei-Yue: Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 49 (2019), Nr. 11, S. 2266–2277

[27] Weathersby, Jason: Linking Algorand Stateful and Stateless Smart Contracts. (2020). – `https://developer.algorand.org/articles/linking-algorand-stateful-and-stateless-smart-contracts/`

[28] Weathersby, Jason: https://developer.algorand.org/articles/using-a-smart-contract-to-spawn-additional-smart-contracts-2/. (2022)

[29] Weathersby, Jason: *Contract to Contract calls and an ABI come to Algorand*. 2022, February 25. – `https://developer.algorand.org/articles/contract-to-contract-calls-and-an-abi-come-to-algorand/`

[30] Zheng, Zibin ; Xie, Shaoan ; Dai, Hongning ; Chen, Xiangping ; Wang, Huaimin: An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. In: *2017 IEEE International Congress on Big Data (BigData Congress)*, 2017, S. 557–564

List of Figures

List of Tables

Statement of authorship

I hereby certify that I have authored this document entitled *Contract to Contract Calls for Financial Applications in Algorand* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, August 17, 2022

Vasil Petrov