Migrating Smart Contracts Across Heterogeneous Blockchains

Luís Miguel de Castro Abrunhosa

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. André Ferreira Ferrão Couto e Vasconcelos
Prof. João Fernando Peixoto Ferreira

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. André Ferreira Ferrão Couto e Vasconcelos
Member of the Committee: Prof. Sérgio Luís Proença Duarte Guerreiro

November 2021

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thank my aunts, uncles and cousins for their understanding and support throughout all these years. I would like to thank to my closest friends for helping me in good and bad times in my life.

I would also like to acknowledge my dissertation supervisors Prof. André Vasconcelos and Prof. João Ferreira for their insight, support and sharing of knowledge that has made this Thesis possible.

Last I would like to thank Rafael Belchior for his guidance as well, and for his time spent helping me to take out the best version of this work, along with my supervisors.

To each and every one of you – Thank you.

Abstract

Migration is an important topic of blockchain technology. Once a blockchain becomes obsolete, or another one emerges with new and more appealing features, it is necessary to migrate all the data, including their smart contracts. Smart contracts are a way of users to establish communication/transactions between each other. We present *Osprey*, a smart contract migration tool between heterogeneous blockchains.

Osprey is a flexible tool integrated as a *Hyperledger Cactus* plugin, that allows the translation of *Solidity* smart contracts into *Typescript Hyperledger Fabric chaincode*. *Osprey* was tested on a curated dataset of 13 Solidity smart contracts and takes on average 3.68 milliseconds to translate them. Also, we conducted a survey where on average, *Osprey* ranked as a moderated structured and readable translated tool.

Keywords

migration, translation, blockchain, hyperledger fabric, chaincode, smart contracts, ethereum, solidity.

Resumo

A migração é um tópico importante na tecnologia blockchain. Uma vez que uma blockchain fica obsoleta, ou outra emerge com novas funcionalidades, é necessário migrar toda a informação, incluíndo os smart contracts. Smart contracts são o que permite os utilizadores estabelecerem transações entre si.

Apresentamos *Osprey*, uma ferramenta flexível e integrada como um plugin para o *Hyperledger Cactus*, que permite a migração de smart contracts escritos em *Solidity* para *Hyperledger Fabric chaincodes* escritos em *Typescript*. No processo de tradução do dataset contendo 13 smart contracts, *Osprey* apresentou em média 3,68 milissegundos, no processo de tradução do seu dataset com 13 smart contracts. Mais ainda, numa survey realizada, em média, *Osprey* foi classificada como sendo uma ferramenta, cuja tradução é razoavelmente estruturada e percetível quanto à leitura do código.

Palavras Chave

migração, tradução, blockchain, hyperledger fabric, chaincode, smart contracts, ethereum, solidity.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Introduction

Contents

Blockchain technology has grown over the decade [3] and drawn the attention in several areas: audits [4], health care [5], education [6], among others. It is a decentralized system, offering privacy, security, transparency, and data immutability [7].

There are many blockchain platforms available, each with its own set of features. Having multiple blockchains gives developers the freedom to choose the blockchain that fulfills their requirements. However, companies should be careful in selecting the blockchain platform on which they develop their applications. If later they find out that it does not meet the requirements of the application, it can be hard to migrate into a different platform, and thus the company may lose monetary resources [8].

Blockchains over the years only focus on surpassing specific obstacles, e.g. more performance, more security [8]. The interoperability scenario between each other is often overlooked [3]. Blockchain interoperability is a novel and an important aspect to consider when choosing a blockchain to start developing. In summary, blockchain interoperability manages all communications between homogeneous (blockchains built under the same virtual machine) and heterogeneous (blockchains built under different virtual machines) blockchains. Thus, leading blockchain technology to raise its adoption and reducing the risks.

Therefore, considering blockchains can become obsolete, it is needed to have some mechanisms to migrate all the information from the obsolete blockchain to newer blockchains [8]. Using the blockchain interoperability mechanism, we make use of *Hyperledger Cactus*, a blockchain connector (explained in Section 2.3.4), that allows to establish a connection between the pair of blockchains in study, *Ethereum* blockchain [9] and *Hyperledger Fabric* [7]. This connection is to achieve the migration of smart contracts from *Ethereum* to *Fabric*, thus providing more flexibility and risk reduction in blockchain technology. Although, data migration has been already conceptualized and is being implemented towards its execution with *Hyperledger Cactus* open source project [10], all work done around smart contract migration is theoretical [11]. It is an important step towards the adoption of blockchain technology and the reduction of risks mentioned early.

We propose a tool converts *Solidity* smart contracts into *Hyperledger Fabric* chaincode. It is based on a parser [12] that extracts information from the *Solidity* smart contracts, and through a converter, it converts that information into *Hyperledger Fabric* chaincode.

1.1 Work Objectives

The primary goal of this study is to develop a tool for migrating smart contracts between heterogeneous blockchains. The objectives of developing this tool are to:

1. Formalize the problem of migrating smart contracts:

   (a) Smart contract data extraction;

   (b) Conversion of that data into the target blockchain smart contract language;
   (c) Guarantee that the behavior in the target contract is equal to the source contract.

2. Propose a smart contract migration tool;
3. Implement in the proposed framework mechanisms that allow the migration of *Solidity* smart contracts to *Fabric* chaincode;

We envisage this study to be widely applicable and help enterprises reduce the effort involved in migrating their existing smart contracts to newer and more appealing blockchains. However, one requirement to use our migration tool is that the smart contract to be migrated has to be written in the *Solidity* program language. The blockchains that use this language to program smart contracts are, e.g. *Ethereum*, *Hyperledger Besu*, and *Quorum*.

Our migration tool will try to replicate the behavior that the *Solidity* smart contract had in its blockchain.

1.2 Document Structure

In this document, we will first introduce an overview of blockchain technology and the way it runs. After that, we focus on some specific blockchains, *Ethereum* and *Hyperledger Fabric*, the way they run, their characteristics and, also some keywords and features their smart contracts have. Next we give an overview about *Hyperledger Besu* and state some differences between *Besu* and *Ethereum* blockchain. After that, we present Cross-Blockchain Communication, Blockchain Interoperability, where we show the categories and protocols to establish communication between blockchains and, where we present *Hyperledger Cactus*. Lastly, we present an overview of compilers and some patterns used in blockchain migrators. It will help the reader understand the following chapters.

In Chapter 3, we show some related work about this topic of migrating smart contracts between heterogeneous blockchains, some familiarities, and differences they have compared with our tool.

Following, we present *Osprey*, the tool we developed to migrate smart contracts between heterogeneous blockchains. We also explain its architecture, its modules, and some decisions made during the implementation process.

In Chapter 5, we present the evaluation process of the tool and some conclusions about the translations process based on that evaluation. Moreover, we show some results obtained from a survey where we tried to get some feedback on the translation process, its readability, and structure.

Concluding, we present the conclusions gotten from the development of the tool. The contributions to blockchain technology, and the work that can be made in the future, not only improve the tool as a smart contract blockchain migration but also allow the translation process to be more efficient and flexible.

2

Background

Contents

In this section, we give an overview of blockchain technology. Blockchain technology can be interpreted as a distributed system where its network peers are trustless. A trustless environment in which every decision-making is done by agreement between all peers, eradicating the centralized paradigm. After that, we introduce Hyperledger Fabric [1], Ethereum [9], Hyperledger Besu [13] and Hyperledger Cactus [10]. Thenceforth, the smart contracts and their migration between heterogeneous blockchain. To further understand the concept of heterogeneous blockchain, we discuss blockchain interoperability and how blockchains communicate with each other. Given the project's context and the envisaged case study, the focus is on permissioned blockchains. Furthermore, to complement our study, we will discuss some aspects of blockchain migration [14].

2.1 An Introduction to Blockchain

A blockchain is a decentralized, distributed system composed of multiple machines. Each machine contributes to computational and storage resources, known as nodes. Nodes are considered to be untrusted. Thus, the decision power is divided between them. This untrusted behavior is part of a model followed by most blockchains, the *Byzantine Failure model* [7]. This model describes that every node can be faulty, including malicious nodes. A faulty node is a node that crashes or has an inconsistent state. It will agree on some states differently than other nodes. To tackle this issue, blockchains implement a consensus protocol. It declares that all nodes will agree on some state. This state is replicated in every node. In other words, all nodes share and update this state, known as the global state or distributed ledger. A distributed ledger is composed of multiple blocks chained chronologically, where each block contains the blockchain's global state and information about the next node in the chain.

A transaction, in blockchain technology, is the representation of an interaction between two parties. This interaction can be a simple exchange of digital coins (cryptocurrency) or could be a simple file transfer. Additionally, users create transactions for two purposes: (1) deploy functions and state, expressed as code in a smart contract to be consumed by other users; (2) provide their data to trigger some service to be used. Moreover, blockchain technology is immutable, because the information that changes the state of the blockchain cannot be changed only appended, tamper-evident and tamper-resistant. After all, to the information being stored in the distributed ledger, it should first be validated by each participant of the network. The security and tamper evidence characteristics are guaranteed through cryptographic signatures and using hash functions. Tamper-resistant, through the way the state is persisted (using Merkle trees, where each node in the tree is a hash of its children). The cryptographic signature allows proving the data provenance in a transaction. The hash function lets data be efficiently verified. Also, it represents the output state from the smart contract's execution.

A smart contract is a collection of functions deployed in the blockchain that use the user's data to perform a service. Also, in a smart contract, its functions must be deterministic, otherwise, it produces a runtime error. Furthermore, the smart contract language can vary from blockchain to blockchain, e.g. in *Ethereum*, the smart contracts are written in *Solidity*, as for *Hyperledger Fabric* the smart contracts can be in *TypeScript*. As mentioned before, a transfer can be a simple exchange of cryptocurrencies. However, not all blockchains have cryptocurrencies. The cryptocurrencies are more often used in permissionless blockchains than in permissioned blockchains (e.g. *Bitcoin* [15]).

Permissionless blockchains [3] are blockchains that do not require authorization or permission to participate in the network, e.g. *Ethereum*. This kind of blockchain use incentives to maintain trust in the system. Thus, tackling most of the nodes' malicious behavior. Furthermore, these incentives motivate nodes to produce blocks and contribute to the network. The production of blocks, e.g. in Bitcoin, follows a consensus called *Proof of Work* [16], which is all the peers competing with each other to see which one is faster enough to be the first one to solve very complex cryptographically puzzles to build a block. Once a node solves that puzzle, it shares it with all peers so they can append the block to their ledger. Contrary, a permissioned blockchain is a restricted one, where it needs authorization and permission to be part of the network, e.g. *Hyperledger Fabric*. In this kind of blockchain, the production of blocks, or the mining process, is not built on incentives but in a modular consensus based on endorsement policies enforced by the network administrator [1].

Apart from permission or permissionless environment, blockchains in their runtime can be homogeneous and heterogeneous.

Homogeneous blockchains share compatibility in their runtime, meaning their contracts can run on the same execution environment. Contrarily, heterogeneous blockchains do not share compatibility, meaning contracts run on their blockchain's runtime.

2.1.1 Ethereum Blockchain

Ethereum is a public blockchain that allows anyone to participate in it. The participants are called nodes. Also, *Ethereum* built a virtual machine, *Ethereum Virtual Machine* (EVM), to operate in their environment and create a programming language to program its smart contracts. These two features, EVM and the creation of a programming language, allowed *Ethereum* to be more efficient and faster than its competitor, *Bitcoin* blockchain. In *Bitcoin*, developers only use predefined functions, being more limited in specifying the behavior of their smart contract. Thus, being vulnerable to attacks in case the default implementation of the functions presents any vulnerabilities.

Ethereum smart contracts are written in *Solidity*. *Solidity* is a program language influenced by *JavaScript*, *Python*, and *C++* [17]. As mentioned, different from *Bitcoin*, in *Solidity* developers can make condition to limit the terms of using their smart contracts, building a defense from attackers in the smart contract. Also, *Solidity* is considered a *Turing-complete* programming language, meaning developers can write their own rules and conditions in smart contracts.

2.1.1.A Programming in Ethereum

In *Ethereum* blockchain the most popular programming language to write smart contracts is *Solidity*. Comparing with its competitor *Bitcoin*, *Ethereum* has improved by allowing developers to define their owned rules and state in the smart contract. It means that before any smart contract is executed, some conditions enforced by the blockchain itself must be met [9]. Smart contract execution is triggered by users. To trigger a smart contract function, a user must have an account (a wallet). The wallet is where it contains the keys (public and private keys) to sign transactions. Also, int the wallet is where it is stored all the amount in ether (*Ethereum* cryptocurrency) of the user. The information about the user who triggered the transaction is obtained, through the **keyword** msg. This **keyword** shows the information about the user who triggered the smart contract and, it has, among others, the following properties:

- sender - the identity of the user triggered the smart contract transaction.
- value - the amount of money the user wants to send in the transaction.

Furthermore, the wallets in *Ethereum* have an address associated. Every transaction is triggered, by a user who has a wallet address associated. To this address, there is a function that is essential to transacting money between users. This function is the send function. The send function takes the wallet address where the money will be transferred and, the function's parameter is the amount to be deposit. Although in Listing 2.1 we show a smart contract in *Solidity* which is using the function transfer, *Solidity* documentation [17] specifically mentions that this function must be used with caution, due to the vulnerabilities that can be exploited by attackers through that function.

Regarding the store of data, *Solidity* offers three types of space to store data [17]: stack, memory, and storage. Storage is a key-value store, where data persists between function calls and transactions. The memory area is similar to the storage area. However, instead of data being persisted between function calls and transactions, the contract gets new instances for every function call. Last, the stack is where all computational operations are stored.

At last, there are some other keywords to be aware of when implementing a smart contract in *Solidity*. The first one is the payable keyword this keyword signals the compiler, that in that specific function that is being triggered, there is some fee to be paid by the user who triggered. The second one is the modifier keyword, which is the conditions that developers enforce to allow the function to be executed. If any of the conditions the modifiers are enforced fail, the smart contract's function is not executed and the transaction fails.

In Listing 2.1 shows a simple smart contract written in *Solidity*, taken from the smartbugs dataset [18, 19], that exposes only one function to send money between users.

Listing 2.1: Hello World *Solidity* smart contract from smartbugs dataset [2]

```
pragma solidity >=0.4.22 <0.8.0;
pragma solidity ^0.4.24;

contract MyContract {
    address owner;
    function MyContract() public {
        owner = msg.sender;
    }
    function sendTo(address receiver, uint amount) public {
        // <yes> <report> ACCESS_CONTROL
        require(tx.origin == owner);
        receiver.transfer(amount);
    }
}
```

2.1.1.B Ethereum Interaction: User - Smart Contract - Blockchain

Since the *Ethereum* blockchain is also a decentralized system, it has a consensus protocol. The consensus protocol used by *Ethereum* is the Proof of Work (PoW). To prevent attacks, such as *Denial of Service* (DoS), *Ethereum* uses the concept of gas [9]. *Gas* is an amount of digital coin a user must pay according to the number of operations, the smart contract that he uses to perform. *Ethereum*'s a digital currency, or cryptocurrency is called Ether. Following this concept, trying to perform such an attack as DoS, comes with huge expenses. Thus, with this approach, *Ethereum* guarantees the correct execution of all the intervenients in the network. Users pay Ether to *miners*, so they validate, build, and store the data transferred in the blockchain. A *Miner* is a node in the network whose job is to validate the network itself. Thus, they receive, propagate, verify, and execute transactions crossing across the blockchain. When the execution finishes, they group transactions into blocks and update the state of the blockchain.

2.1.2 Hyperledger Fabric

Hyperledger Fabric is a distributed operating system for permissioned blockchain able to run distributed applications written in *Java*, *Go* and *NodeJs* (*TypeScript*). It traces the execution history, securely, in a replicated ledger and has no cryptocurrencies associated. Unlike most public blockchains which use order-execute aproach, Fabric introduces a new approach, called order-execute-validate. In Fabric a distributed application is composed by two parts:

- A smart contract or a *chaincode*, where the application's logic is implemented. It is executed in the execution phase. In *Fabric*, *chaincodes* are the core of distributed applications. *System chaincodes* are special *chaincodes* responsible to manage the blockchain.

- An *endorsement policy* used in validation phase. *Endorsement policies* serve to validate a transaction and are parameterized by the *chaincode*. They choose a set of nodes to validate the transactions, called *endorsers*. Also, only administrators can change them.

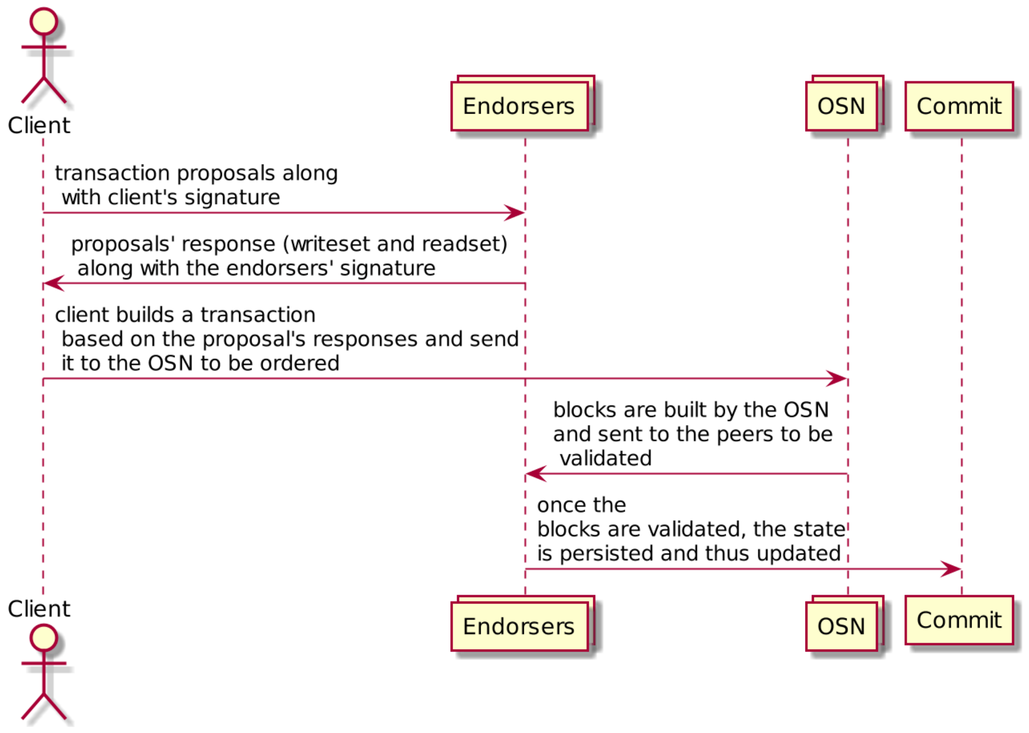

Client
Endorsers
OSN
Commit
transaction proposals along with client's signature
proposals' response (writeset and readset) along with the endorsers' signature
client builds a transaction based on the proposal's responses and send it to the OSN to be ordered
blocks are built by the OSN and sent to the peers to be validated
once the blocks are validated, the state is persisted and thus updated
Client
Endorsers
OSN
Commit

Figure 2.1: Execute-Order-Validate Approach, adapted from Androulaki's paper [1]

In Figure 2.1, is shown a sequence flow of the execute-order-validate approach. *Endorsement policies* specify which peers must execute and store the output of each transaction sent by clients. This process is the *endorsement phase*. After that comes the *ordering phase*. It uses a consensus protocol to order the endorsed transactions grouped in blocks. Then, it broadcasts the blocks to all peers. This phase may use the *gossip* protocol. Fabric orders the transaction's output fused with state dependencies processed in *execution phase*. Then, all peers validate the endorsed transactions and in the same order. The *validation phase* is deterministic. With this approach, Fabric introduces a hybrid replication in the Byzantine model, which puts together the passive and active replications.

A blockchain has a set of peers that form a network. These peers must have the authorization to participate because Fabric is a permissioned blockchain and an identity provided by an MSP (Membership Service Provider). Each peer can have at least one of the following roles in the network:

- *Clients* present transactions for execution, help in the execution phase, and broadcast those transactions for the ordering phase.
- *Peers* execute and validate transactions. They maintain a blockchain ledger, a data structure in the form of an hash chain, where all transactions and state are stored. The state is the latest ledger state. Only the endorsers or endorsing peers, a subset of all peers specified by a policy of the chaincode where the transaction belongs, execute the transactions in the execution phase.
- *Ordering Service Nodes (OSN) or the orderers* are the peers working together to form the ordering service. This service total orders all transactions. Each transaction contains state updates and dependencies computed in the *execution phase*. They also have the cryptographic signatures of the endorsers. Orderers do not know the state of the application and do not integrate the *execution phase* nor the *validation phase*.

Execution Phase, clients send transaction proposals signed to one or multiple endorsers. Each *chaincode* specify, through the *endorserment policy*, which peers may execute those proposals. When arrived, the endorsers simulate them, on a Docker, in the specified chaincode and apart from the main process. The result of the simulations does not persist in the ledger state. Furthermore, the Peers Transaction Manager (PTM), a versioned key-value store, maintain the blockchain state. The state created by a particular *chaincode* cannot be accessed by another *chaincode* unless it has permissions to invoke it in the same channel. It is a way of a specific *chaincode* to read another's chaincode state. The results of the simulation are a writeset and a readset. The writeset represents the state updates, the simulation outputs. The readset represents the proposal simulation version dependencies. This result is then signed, cryptographically, by the endorsers and sent to the client as a proposal response or endorsement. After the client receives all the endorsements specified by the chaincode's endorsement policy, he verifies that all the responses are equal. After that, he produces a transaction and sends it to the ordering service.

Ordering Phase happens after the client gets the number of transaction proposals established by the policy. It creates a transaction and sends it to the ordering service, called the *ordering phase*. This phase takes the transactions submitted and orders them atomically to ensure the consensus protocol in each transaction. Furthermore, the ordering service puts a set of transactions into blocks and outputs an hash-chain of those blocks. This way, it is easier and efficient to validate the blocks later.

This service offers two operations, broadcast and deliver. The first operation enables users to send a specific transaction to the network. As for the other operation, allow peers to receive transactions from the network. Also, the ordering service guarantees that all blocks received per channel are totally ordered. For each channel, this ordering service ensures the following safety properties:

- *Agreement*: ensures that non-faulty peers agreed on the same output.

- *Hash-chain integrity*: ensures if two blocks are delivered by correct nodes, one with a number s and another with a number s+1, then both blocks have the same hash-chain.
- *No skipping*: if a non-faulty peer delivers a block with some number s ¿ 0, then all blocks with the number s' ¡ s, were sent.
- *No creation*: if non-faulty peer delivers some block with number s, then all transactions of that block are already broadcasted.

For liveness, this service offers at least this “eventual” property:

- *Validity:* if a non-faulty client broadcasts a transaction, then eventually all non-faulty peers will deliver that transaction with some sequence number.

Validation Phase consists of receiving blocks from the ordering service or through the *gossip* protocol. When arrived, this phase divides into three steps:

1. *Endorsement policy evaluation* executed in parallel for all transactions in the block. This evaluation is also called validation system chaincode (VSCC). It is responsible for validating if the endorsements follow the endorsement policies specified by each *chaincode*.
2. *Read-Write conflict check* is done to all transactions in the block sequentially to ensure the local state of the endorser (before sending the response proposal) is still the same as the ones received. It uses the version of the keys in the readset to compare with those maintained in the endorser's local state.
3. *Ledger update phase* is when all the valid transactions in the block are committed to the ledger and so, update the blockchain state.

2.1.2.A Programming in Hyperledger Fabric

Hyperledger Fabric chaincodes can be developed, as mentioned before, with four programming languages: *Java*, *Typescript*, *Go* and *Javascript*. Similar to *Solidity*, in *Typescript*, *Hyperledger Fabric* framework has some specific objects that makes the network know the entry point of each *chaincode*. To identify a *chaincode*, the main class where all the functions to be exposed to the network are, must extend a class called *Contract*. Also, every function, to be exposed to the network, of the class that extends the *Contract*, must always have as its first parameter, a *Context*. The *Context* object is another object of the *Fabric* framework injected automatically in the function when they are triggered by users. It contains all the information about the user who triggered the *chaincode*. Bridging to *Solidity*, is similar to the object *msg*.

Different from *Ethereum* blockchain, where the global state is managed and organized by the framework itself, in *Hyperledger Fabric* there is no notion of global state. The global state is a key-value pair storage. Also, is in the *chaincode* where the developers decide what is going to be stored as state and how (e.g. what will be the key value and, which values will be associated).

In Listing 2.2, it is the Hello World *Solidity* smart contract translated to *Javascript* using a tool called *solidity2chaincode* [20].

Listing 2.2: Hello World chaincode traduzido para Javascript usando a ferramenta solidity2chaincode [20]

```
const ClientIdentity = require('fabric-shim').ClientIdentity;
class MyContract {
    async owner(stub, args, thisClass) {
        let tmp = await stub.getState('owner');
        return Buffer.from(tmp.toString());
    }
    async Constructor(stub, args, thisClass) {
        let owner = new ClientIdentity(stub).getID();
        await stub.putState('owner', Buffer.from(owner.toString()));
    }
    async sendTo(stub, msg, receiver, amount) {
        let txOrigin = new ClientIdentity(stub).getID();
        if(txOrigin != msg.sender){
            throw new Error("Only owner can call this.");
        }
        if (msg.value > amount) {
            let args = ['send', msg.sender, receiver, amount.toString()];
            await stub.invokeChaincode('balance', args);
            msg.value = msg.value - amount;
        } else {
            throw new Error('Exception during transfer');
        }
    }
}
```

Comparing Listing 2.2 with 2.1, we can see some similarities. To users trigger transactions in the blockchain, the smart contract must expose functions for users to use.

In terms of differences, *Solidity* is a typified programming language, while *Javascript* is a dynamic language. It means, in *Solidity*, to declare a variable, the type must be explicit, or when to assign a value to a variable, both must have the same type, or a compile error is thrown. *Javascript*, on the other hand, there is no notion of variable type. Thus a variable that contains a number value can change to a string value, no errors are thrown.

2.1.3 Hyperledger Besu

Hyperledger Besu is an Ethereum client licensed under Apache 2.0 [13]. It is an open-source project written in *Java* and can be executed in *Ethereum* public or private network. Also, it can be run in tests environments such as *Rinkeby*, *Ropsten* or *Görli*. Besides *Hyperledger Besu* be an *Ethereum* client, it supports some *Ethereum* functionalities such as Ether mining, Smart contract development, and Decentralized applications development. *Hyperledger Besu* implements Ethash Proof of Work and IBFT 2.0 and Clique Proof of Authority as its consensus protocol.

Rinkeby and private networks use *Clique Proof of Authority*. It is composed of approved accounts (signers) whose role is to validate transactions and validate blocks. Additionally, it is a set of Signers that produce the blocks. They do it in turns. For instance, if Signer D produces block A, the next block B will be produced by Signer E, and so on. Furthermore, Signers can vote to add or remove other or new Signers to the network.

Private networks use *IBFT 2.0 Proof of Authority*. It has the same attributes as Clique Proof of Authority. However, in Clique the approved accounts are denominated as Signers. In IBFT 2.0 they are denominated as validators. The approved accounts in both Proofs of Authority algorithms have the same roles. However, in IBFT 2.0 the blocks must have the signature of at least 66% of the validators to produce.

2.1.3.A Hyperledger Besu vs Ethereum

As mentioned, *Hyperledger Besu* is an *Ethereum* client project. It exposes command line functions and a *JSON-RPC API* that enables users to run, maintain, debug and monitor nodes in the *Ethereum* network.

Hyperledger Besu allows users to mine ether, develop smart contracts and decentralized applications. Also, *Besu* supports the deployment of smart contracts. However, contrarily to *Ethereum*, *Besu* does not support key management.

2.2 Cross-Blockchain Communication

Cross-Blockchain Communication is the process of a source blockchain establishing communication with a target blockchain to exchange transactions [3]. Each transaction exchanged is instantiated in the source blockchain and then executed in the target blockchain. Also, this process of establishing a communication between blockchains uses two communication concepts, the *cross-chain communication protocol* (CCCP) and *cross-blockchain communication protocol* (CBCP).

Cross-Blockchain Communication is important to blockchain migration because it leverages all communication aspects between both blockchains (the source and the target). The CCCP and CBCP protocols are useful in terms of accessing a specific source blockchain to obtain information, such as blockchain data, smart contracts. This information is processed and then, deployed on the target blockchain. All communication details such as compatibility, synchronization of the transaction are leveraged by those protocols, resulting in successful blockchain interoperability.

2.3 Blockchain Interoperability

Blockchain Interoperability is a technique that allows homogeneous and heterogenous blockchains to coexist. This coexistence means the range of blockchains involved can communicate or even complement each other, not only for migration purposes (e.g. when the source becomes obsolete) but for business purposes (e.g. running the same kind of business in a different blockchain).

Blockchain interoperability is crucial for the migration of data or smart contracts because per se blockchain interoperability is connected with the communication establish between blockchains, being compatible or not in its core. If we want to simply establish a connection between *Hyperledger Fabric* and *Ethereum*, the protocols used and techniques are all compiled in the essence of interoperability between blockchains.

Blockchain interoperability can be divided into three categories: *Cryptocurrency-based approaches*, *Blockchain Connectors*, *Blockchain Engines* [14]. Our study will be focused mostly on *Blockchain Connectors*, specifically *Blockchain Migration*.

2.3.1 Cryptocurrency-Based Approaches

Cryptocurrency-Based Approaches is a strategy that can be divided into four solutions. These solutions approach the interoperability between chains and the way they cooperate, using mostly cryptocurrencies. The four solutions are the sidechain (or relay chain) approach, notary schemes, timed hash-locks, and hybrid solutions.

2.3.1.A Sidechain

Sidechain is a system composed of two or more blockchains. This blockchains can have three possible roles, be a *mainchain*, *sidechain* or both (*mainchain* and *sidechain*) ([21–25]). *Mainchains* are blockchains used, mainly as the primary storage, but also as the primary system. *Sidechains* are blockchains used, mainly as an extension of the *mainchain*, primarily for storage purposes. A use case for this approach is, the blockchain has a large number of users (i.e. in Fabric), and does not have enough storage resources to record all the users' data.

Sidechains are mostly used for transferring assets. This transferring process uses the two-way peg mechanism. This mechanism locks the number of assets transferred in the source blockchain and registers it in the target blockchain.

In the context of migration, this approach can be used to leverage the incompatibility between newly developed blockchains with others that already exist. Let Blockchain A be a newly developed blockchain technology that was based on the *Ethereum* blockchain but without the need of having cryptocurrencies to run the system. Moreover, this blockchain A brought new unique features that revolutionize blockchain technology. Although the compatibility with *Ethereum* blockchain is high, compared with heterogeneous blockchains, there is no compatibility. In this case, a sidechain can be used to make the migration process, being the *Ethereum* blockchain a b́ypass" of information between blockchains.

2.3.1.B Notary Schemes

Notary Schemes are third-parties that monitor all the transactions triggered in the various blockchains ([26,27]). Those third parties can be centralized or decentralized. Additionally, Notary schemes perform operations over the blockchain to validate the transactions (i.e. cryptocurrency exchange), rather than being an extension of the blockchain like the sidechain solution. This approach can be used for migration to validate the behavior of the data that is being migrated to the target blockchain.

2.3.1.C Hashed Time-Locks

Hashed Time-Locks is a decentralized way to make exchanges between blockchains [28]. It uses hashes and timelocks to ensure atomicity in all operations. The hash proves the validity of transactions. The timelocks ensure that the cryptographic proof is delivered in a specific interval of time. This technique enables blockchains to perform *atomic swaps* in transactions. An *Atomic swap* allows users to exchange cryptocurrencies between blockchains. In terms of migration purposes, this approach can be used to protect the data being migrated to the target blockchain, so a malicious node does not perform an attack to tamper with the information that is being migrated.

2.3.1.D Hybrid solutions

Hybrid solutions are a concept of interoperability that takes care of the users' private key distribution [29–32]. Their incompatibility in the target blockchain makes them generate new key pairs. Additionally, the generation of key pairs makes blockchains associate the old key pairs with the new ones. Moreover, hybrid solutions decentralize the management of assets between several nodes, allowing developers to implement decentralized notary schemes and two-way peg mechanisms. This approach can be used to link wallets between blockchains in the migration process. This link will allow the transfer of assets to be done successfully because the old key pairs of the source blockchain are pointed to the newly generated in the target blockchain.

2.3.2 Blockchain Engines

Blockchain Engines focus mostly on heterogeneous blockchains. They are frameworks that reuse their components (network, consensus, contract, incentive, and data components) to customize the blockchain. Blockchain Engines allow developers to build applications (or decentralized applications) and specify the transferring of assets and information between blockchains without specifying the blockchain. Furthermore, Blockchain Engines assure blockchain instances beforehand, misconcerning developers of their creation.

Blockchain Engines in terms of migration is an abstraction of the whole process of migrating information between blockchains. It allows developers to write their smart contracts in one specific programming language and, when deploying that smart contract, it will make the code compilation of the target blockchain where that smart contract will run.

2.3.3 Blockchain Connectors

Blockchain Connectors are a set of implementation concepts that allow interoperability between blockchains at a low-level. This solution can be divided into 4 sub-categories, *Trusted Relays*, *Blockchain Agnostic Protocol*, *Blockchain of Blockchains* and *Blockchain Migrators*.

Trusted Relays are a routing table third party that behaves like a proxy. It knows all the addresses in the network to redirect the transactions. *Blockchian Agnostic Protocol*, is a solution that enables communications between heterogeneous and homogeneous blockchains. *Blockchain of Blockchains* is a solution that maintains blocks that contain all the information about the blockchains in the system. *Blockchain Migrators* will be addressed in Section 2.5.

Blockchain Connectors are a useful solution in the migration process because it leverages all the dependencies that a specific migration tool must-have. These dependencies materialize in the implementation regarding the validations [33] and connections to the blockchains. *Blockchain Connectors* this way can be a generic module that can be used, despite the blockchain we want to migrate to.

2.3.4 Hyperledger Cactus

Hyperledger Cactus is a Blockchain Connector, more specific, a *Trusted Relay* implementation, discussed in Section 2.3.3. This connector's job is to guarantee interoperability [14] between cross-chain transactions. Both concepts, interoperability and cross-chain transactions, will be explain in Sections 2.3 and 2.2, respectively.

Hyperledger Cactus is composed of nodes. Each node has a connector, a validator, and a set of plugins manage by the system administrator [34]. Also, *Cactus* nodes are a plugin-based architecture that gives great modularity. This modularity is because the administrator can add or remove the plugins according to their interest. Additionally, it is an API server that provides these plugins to the *Cactus* node. The connector is responsible for establishing communications with the blockchains, both source, and target. The validator takes the transactions and checks their validity.

Regarding the *Cactus* compatibility, it is compatible with most *hyperledger* technologies, and it is under developement the integration of *Cactus* with public blockchains.

2.3.4.A Architecture

Hyperledger Cactus is a plugin-based architecture characterized by being flexible in terms of extension to new features. This flexibility is mostly achieved due to the way it was designed. Its design is based on an interface based programming, where it leverages any kind of implementation dependency. In order to implement a new feature, a dependency to the cactus plugin core or to other plugin in which we desired to consume, must be added to the file of dependencies (package.json). All plugins are located in the packages directory. From the project root directory is where we add our tool as a cactus plugin. To make our tool become a plugin, we need to insert also the dependency to the package.json file that is located in the project root directory.

Following the design and structure of the *Hyperledger Cactus* creator, each plugin must be structure with a source file. The source folder is where all the project implementation and testing is located.

Going deeper in the source directory, we have two main directories, a main directory and a test directory. The main directory is where all the plugin logic is implemented and where we expose the functions to be used not only to the front-end if we use as a web application, but also to be consumed by other plugins that may want to use. The second folder is the test directory. In this directory is where all plugin tests are made. It is here where developers test not only the behavior of each function to be exposed to the outside world, but also in integration with other dependencies that the plugin might have.

2.4 An Overview about Compiler

Compilers primarily analyze the program code and output a newly generated code that a determined machine or program can interpret and execute [35]. That analysis can be divided into three phases: Parsing, Transformation, and Code Generation. To better understand each phase of the compiler we will break down an example of having a *Ethereum* smart contract written in *Solidity* and turn it into a *Hyperledger Fabric Typescript chaincode* (Listing A.3).

The Parsing phase is responsible to analyse the smart contract file and, break down into two phases. The first phase is the Lexical Analysis and the latter one is the Syntactic Analysis. Lexical Analysis is the process of extracting numbers, labels, punctuation, operators, among others. Syntactic Analysis is the process of taking all the information taken from the Lexical phase and build a representation of that data. That representation is called *Abstract Syntax Tree* or AST. An AST is a representation of the code that is better understood, in terms of information. Listing A.4 shows us an example of the AST extracted from the code of Listing A.3.

The Transformation phase is where the compiler will make some changes over the generated AST. An AST is composed of nodes. Each node tells information of the data extracted from the smart contract file. In this phase, the changes are made in each node, or it can be a newly generated *Abstract Syntax Tree*. In Listing A.4, the visibility and constructor properties can be a modification on the AST node, so the node has additional information that helps the next phase of the compiler.

The Code Generation is the final phase of a compiler. In this phase compilers can generate new information that may overlap the information added in the transformation phase, others can just stringify and return the generated AST without making any changes.

2.5 Blockchain Migrators

Blockchain Migrators are the process of users to migrate their blockchain information (i.e. storage, state, smart contracts) to a different blockchain [3, 14]. Although this is a big step in blockchain interoperability, some proposed solutions aim to turn this concept into reality. For instance, in smart contracts' migration, there are patterns [8] that can be used to build a smart contract migration tool. This patterns are *Virtual Machine Emulation Pattern* and *Smart Contract Translation Pattern*.

Smart Contract Migration Pattern are a set of steps, designed to allow smart contracts to run on different kinds of blockchains, whether homogeneous or heterogeneous.

Virtual Machine Emulation Pattern allows a company to migrate the state of a VM as well as the state of smart contracts between blockchains. If their execution environment is the same, then there is no need to perform a copy of the VM from the source blockchain and install it on the target blockchain (in practice, the target blockchain uses a third-party virtual machine to translate the instruction set).

Otherwise, it needs to make a copy and install it. After the installation process of the VM's copy, the token burning pattern is used in the smart contract to be migrated, to destroy it in the source blockchain. So it can't be used anymore. Moreover, to run the smart contract in the target blockchain three steps must happen. First, the deployment of the smart contract in the target blockchain. When deploying the smart contract, it is necessary to set the state of the smart contract, using the state initialization pattern. However, the smart contract' code may not be in the snapshot, so the same pattern is used. After that, the smart contract must be redeployed in the target blockchain. Second, update the smart contract' identifier, since the smart contracts' address may be different across blockchain instances. Third add a proof of existence entry (PoE) of all the ID's updates, made in the database.

Smart Contract Translation Pattern allows, to "automatically" compile a smart contract, with a specific programming language, into another smart contract, with a different programming language, and run in a distinct blockchain. This is possible because for the translation to be made, first, it needs to provide the smart contract's source code to be translated. After the translator got all the information needed, it converts into the new smart contract. After that, it runs some tests to compare the behavior, correctness, safety, and binary of the newly created smart contract. This testing phase is crucial, to generate the same output state from the one generated in the source blockchain. After the testing phase of the new smart contract, the following steps are the same as the previously mentioned pattern, regarding the deployment of the smart contract, setting the state, update the ID database, and adding the Proof of Existence entry of the updated IDs.

3 Related Work

Contents

In this section, we discuss solutions relating to the migration of smart contracts between heterogeneous blockchains. First, we present Hyperservice [11], a framework able to abstract which blockchains are in use. Additionally, the framework has a built-in *Domain Specific language* able to abstract the smart contract program language of each blockchain in it. Next, we discuss a tool [12] able to extract relevant information in smart contracts, specifically *Solidity* smart contracts in *Ethereum* blockchain. Last, we discuss two solutions relating to smart contract migration patterns [8] and the limitations and strengths of using them.

3.1 Hyperservice

Hyperservice is a framework designed to take another step further in blockchain interoperability. It is a platform that helps developers build and execute smart contracts able to run in heterogeneous blockchains [11].

3.1.1 Architecture

Hyperservice is composed of four components: (i) *dApp Clients* which are gateways consuming the services provided by the framework. (ii) *Verifiable Execution Systems* (VESes) are the compilers in the platform that compile decentralized applications into blockchain-executable transactions, Hyperservice executables. (iii) *Network Status Blockchain* (NSB) which is a blockchain of blockchains providing an overview over dApp's execution status. (iv) *Insurance Smart Contract* (ISC) which arbitrate the correctness and violation of dApp's execution in a trust-free manner. Also, ISCs have mechanisms to prevent misbehavior in transactions.

Unified State Model (USM) [11] is, according to the authors, "a blockchain-neutral and extensible model for describing state transitions across different blockchains, which in essential defines cross-chain dApps.". It is accomplished through a *virtualization layer* where unifies the heterogeneous blockchains by including (i) blockchains, regarding its implementations, where it abstracts them through an object containing public state variables and functions. (ii) Developers program dApps only have to specify the operations and the order of them on the objects.

A USM has a set of entities, operations possible to perform over the entities, and the constraints that operations define. Furthermore, there are two types of entities: accounts and contracts. The account is what characterizes a person. It contains its unique address and its account balance. The contract is all the operations and constraints defined (public state variables, callable interfaces, functions, and other attributes) to be executed by clients. Moreover, all entities and operations belong to a local machine, regarding the source blockchain of the smart contract.

Despite operations are local to a machine, when compiled, they eventually result in many transactions on several blockchains. Thus, the synchronization of the consensus processes is not guaranteed. To guarantee this “synchronization” USM establishes some constraints when defining the dependency of operations. There are two kinds of dependencies: preconditions and deadlines. Preconditions are all dependencies satisfied when all the preconditioning operations finish. Thus, with preconditions, developers can order their operations into direct acyclic graphs (DAGs). In these DAGs, the state of the parents of the nodes is persistent. Its children have access to it. Deadlines are all preconditioning operations bounded to a time interval after the dependencies are satisfied. Moreover, with deadlines, applications don’t get stuck and always move forward.

Hyperservice Programming Language (HSL) allows developers to build smart contracts regardless of the cryptocurrency used. Thus, developers can specify through a “universal call-option”, which coin they accept as payment in the smart contract. Additionally, the key aspect of the variety of payment options given to clients is the HSL compiler. HSL compiler is the core of the entire programming framework.

Hyperservice Language Compiler performs two tasks. First, guarantee the security and correctness checks on HSL programs. HSL has a multi-language front-end, based on the source code of the smart contract. It extracts the information of the state variables and functions, then converts it into a USM object. This object passes through serious syntax and correctness checks. Second, compile programs into blockchain-executable transactions. Once the verifications are validated, the compiler generates an executable program. This executable is structured in a *Transaction Dependency Graph*, containing the information about the set of blockchain-executable transactions, metadata of each transaction, the preconditions, and deadlines constraints of the HSL program.

3.1.2 Discussion

Hyperservice solution envisages solving the heterogeneity of blockchains. It abstracts the blockchain layer by having a virtualization layer defining which blockchain the *Unified State Model* will compile. Furthermore, it abstracts the blockchain smart contract language by (i) having in the front-end an interpreter translating the smart contract input. (ii) By developing an hyperservice language and compiler to manage, based on the smart contract, the operations executed on each blockchain. Although Hyperservice is a fined-grain concept towards the interoperability between blockchains and suits all work objectives of this study (1, 2 and 3), it has one limitation. It is a theoretical solution at the time of writing this thesis.

Moreover, Hyperservice is considered as a framework where developers develop under the Hyperservice programming language, and the instructions in the programming file will trigger transaction in the various blockchains that are specified in the code.

3.2 Solidity Parser

Solidity Parser is an open-source translator tool, developed to translate smart contracts in *Solidity* to an *Abstract Syntax Tree* (*AST*) in *Javascript*. The only option to run in another blockchain smart contracts in *Solidity* is through integrating the Ethereum Virtual Machine (EVM) or transcribe the smart contract in *Solidity* to the smart contract language of the target blockchain. Nevertheless, this tool [12], was developed with the goal of translating smart contracts in *Solidity* and allow developers to transcribe them into *chaincode* (*Fabric* smart contract). Also, this parser can “successfully parse up to 75% of the Solidity constructions (types, functions, inheritance, events)”.

3.2.1 Architecture

Translating *Ethereum* smart contracts into *Hyperledger Fabric* smart contracts (*chaincode*) involves two steps: (1) The conceptual mapping of *Ethereum* smart contracts to construct, as much as possible to *Hyperledger Fabric* smart contracts, (2) The development of a source-to-source compiler to maintain the semantic equivalence.

Mapping *Ethereum* to a *Fabric-based* Network. A typical *Ethereum* node maintains its state globally. To be equivalent, *Hyperledger Fabric* has its nodes connected to a single channel. They base the tool on that assumption.

However, contracts in *Ethereum* do not have a notion of version. When instantiating a contract, the actor must specify the name of the *chaincode*. This value (the name of the *chaincode*) is then used to create a contract address and initialize it with a value of zero in the balance of the *chaincode*. If the user’s certificate and contract’s address are strings, then the Ether balance can be stored in it. This is done to keep track of the balance on the accounts and contracts on both blockchains. Moreover, the *chaincode* where the accounts and the balance of the contracts are stored, in Fabric, is called *balance*. This *chaincode* provides functions to send and transfer money (ether) and to query the *balance* of a specific account or contract address. In order, to perform these operations, the X.509 certificate and the name of the contract are checked globally.

To translate smart contracts in *Solidity* into *chaincode*, the tool uses two steps. First is the generation of an Abstract Syntax Tree (AST) in *JSON* format. Second, based on the AST, it performs two iterations over the tree. One to extract, and the other to translate the AST into JavaScript code. The first iteration is to take all the state variables, functions, events, structs, enums, and others. The second iteration is to translate the statements.

Functions, Functions Modifiers, State Variables. *Sol2js* does not make any verification regarding the semantics and syntax of smart contracts in *Solidity*. Thus, each function invocation and modification of state variables are handled based on their visibility (e.g. private functions of a class cannot be accessed in the derived classes). Visibility can be public, external, private, or internal. If everything behaves as expected, then the smart contract is translated without the need of modifying the code.

The tool generates a target function containing a copy of the function modifier along with the function modified. Thus, for state variables with public visibility, it generates a *getter* function that returns the current value of the state variable. Additionally, in the *Hyperledger Fabric*, to store and retrieve state variables, *Sol2js* uses *getState*() and *putState*() functions of *ChaincodeStubInterface*. In the context of the translated code, its size has an impact by the number of state variables contained in the *Solidity* smart contract.

3.2.2 Discussion

Solidity Parser envisages on converting a smart contract in *Solidity* to a smart contract in *Fabric*(1 and 2). It tries to extract all relevant information from the smart contract in *Solidity* (e.g. functions, variables, etc), and *adhoc* transcribe it to *chaincode*. Besides the successful translations are around 75% the tool does not handle some features used in smart contracts in Solidity. *Sol2js* does not support multiple inheritance, function overloading, function types, fixed-point number types, and libraries and type overriding. Also, this tool is not flexible, which means that is a strictly end-to-end translation between *Solidity* and *Javascript*, whereas our solution is more flexible and allows other translations rather than *Solidity* to *Typescript*.

Although our datasets are different, based on the average that the authors presented on the paper [12] (176.72 ms), our solution presented around 3.68 ms. These results show that even our solution does not support some features that *Sol2Js* does, it still is more time performance than *Sol2Js*.

4 Osprey

Contents

4.1 Overview

To explore the possibility of migrating smart contracts between heterogeneous blockchains, we design *Osprey* [36]. We decided to name the migration tool *Osprey*, because the name itself is the name of a migratory bird species and, our goal is to migrate smart contracts from one blockchain to another. *Osprey* is a tool that can translate smart contracts written in *Solidity* to *chaincode* written in *Typescript*. Also, with the integration of *Hyperledger Cactus*, a blockchain connector, *Osprey* can provide a much reliable translation. This happens because *Cactus* provides us a way of instantiating and destroying ledgers, to simulate a running blockchain. These ledgers complement our tool, giving it the ability to prove the behavior of the original smart contract, and the translated *chaincode*. This proof is made, by running the smart contract test files and the translated ones. Thus comparing the result of both executions.

Osprey has a smart contract module and a test module. The smart contract module is responsible to process the input smart contract, pass to the *Abstract Syntax Tree* and then iterate over the tree to translate it into *Typescript chaincode*. The test module is responsible to translate the input *Javascript* unit tests used to test the *Solidity* smart contracts into *Typescript* test files used to test in *Hyperledger Fabric chaincodes*. The way this module works is similar to the smart contract module. It uses an *Abstract Syntax Tree* to make a representation of the unit test file content so it can be interpreted and translated to the structure used in *Typescript* file to test *Hyperledger Fabric chaincodes*.

4.2 Requirements

Although the tool does not address all use cases, meaning it does not perform all kinds of *Solidity* smart contract translations, our goal is to address simple *Solidity* smart contracts that users want to translate to *Typescript Hyperledger Fabric* chaincode.

Osprey was designed with two goals in mind. Translate smart contracts written in *Solidity* to *Typescript chaincode*, and provide flexibility. Flexibility means developers can use our tool to translate smart contracts to other programming languages than *Typescript*, without having to handle the implementation of the intermediate language (*Abstract Syntax Tree*).

4.3 Tool

Hyperlegder Cactus, a blockchain connector, and a plugin-based framework, allow not only to connect with other permissioned blockchains such as *Hyperledger Fabric* but with public blockchains too, such as *Ethereum* blockchain. This connection is achieved through a *Hyperledger* project, called *Hyperledger*

Besu. *Besu* is a *Ethereum* client, that allows performing operations such as get smart contracts, among other operations, in *Ethereum*. This operation is a huge help to our tool because it automatically obtains smart contracts from *Ethereum* and inputs them in *Osprey*. *Osprey* will be a plugin integrated in *Cactus*. After that, the translation process offered by our tool will run, and once it is done, both the input and output will be tested in instances of their blockchains. Once the validations are done, then *Cactus* provide mechanisms to deploy the translated *chaincode* in *Hyperledger Fabric*.

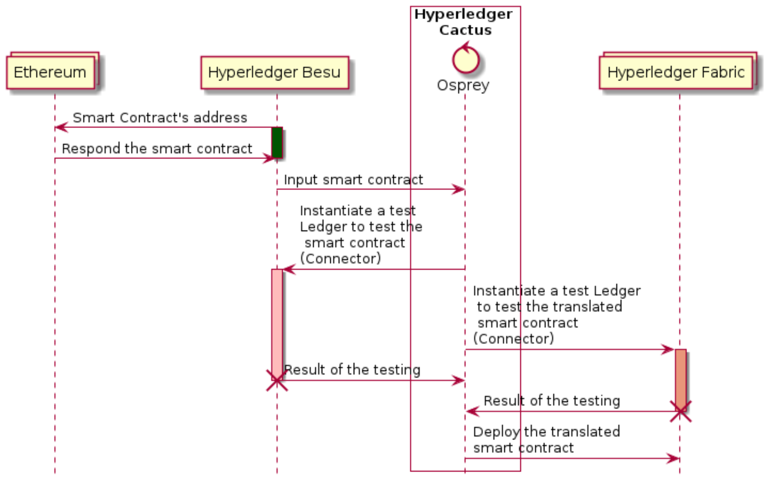

Hyperledger Cactus
Ethereum
Hyperledger Besu
Osprey
Hyperledger Fabric
Smart Contract's address
Respond the smart contract
Input smart contract
Instantiate a test Ledger to test the smart contract (Connector)
Instantiate a test Ledger to test the translated smart contract (Connector)
Result of the testing
Result of the testing
Deploy the translated smart contract

Figure 4.1: Sequence diagram of the translation process integrated with *Hyperledger Cactus*

As Figure 4.1 shows, the first part of the process of migrating smart contracts between heterogeneous blockchains is the retrieving of smart contracts. As explained, this part is done by *Cactus* through establishing a connection with *Hyperledger Besu*. After that, those smart contracts are inputted to *Osprey* and, as you can see in Figure 4.2, the translation of the smart contracts happens. After that, the translated files and the original smart contract files are executed in their ledgers, provided by *Cactus*, to test them. After that, the results of each ledger execution are compared. Once the validation of the executions is successfully checked, the translated *chaincode* is deployed in *Hyperledger Fabric*.

Regarding the translation flow, Figure 4.2 shows us an overview of each step of the process. Once *Besu* returns the smart contract to the connector, the smart contract translation is divided in two ways. The first one is the path of translating the smart contract source files. The ones where the business logic is implemented. The latter one is the path to translate the test files of the project. The ones that prove the business logic is behaving as it supposes to. In section 4.4, we explained in detail, how the translation of the smart contract source files is designed and how it behaves in the translation process. In section 4.5, we explain in detail, how the translation of the smart contract test files is designed and how it behaves in the translation process.

After the process part of the smart contract source files and the smart contract test files are done, then it comes the converting/translating part. this part of the process is where *Osprey* tries to translate

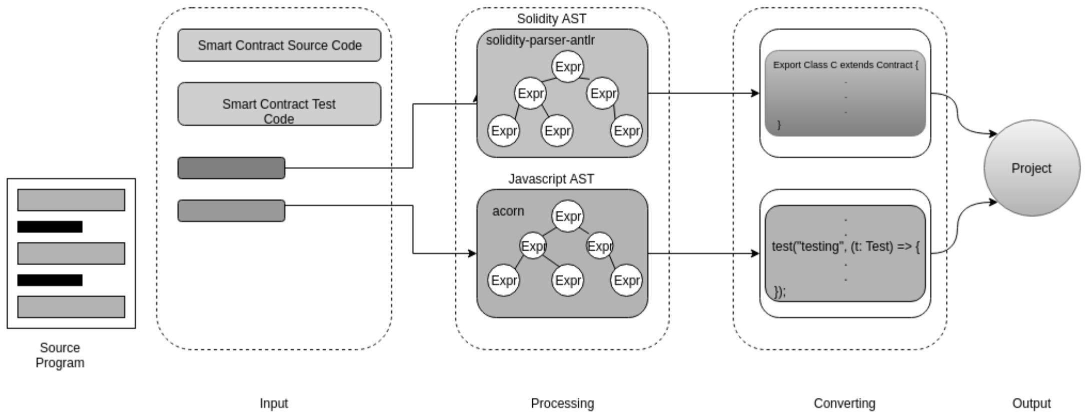


Figure 4.2: Osprey flow overview

the information received from the processing part and write them in the proper files, structuring them into folders, originating the project.

As a *Cactus* plugin, *Osprey* is a microservice tool that can be deployed in the cloud and integrated with blockchain service providers such as azure, aws, among others. The changes to be done to ensure the success of translations are to guarantee that the source blockchain and target blockchain smart contracts programming language is implemented in the tool, otherwise, the tool cannot perform the migration. Figure 4.3 demonstrates the communication.

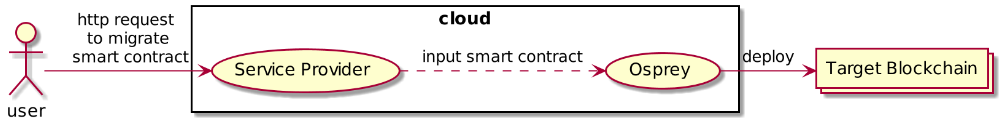


Figure 4.3: Osprey as microservice in the cloud

4.4 Smart Contract Module

As mentioned before, *Osprey* architecture is divided into two modules, the Smart Contract Module, and the Test Module described in Section 4.5. In this section, we explain in detail how this module is structured, what features are implemented, what features are not implemented, and how the translation process behaves.

4.4.1 Architecture

Osprey was designed to be highly interface based, especially in the Test Module (Section 4.5). This decision was made to give developers the freedom to integrate other programming language translations, being those implementations, a plugin. Regarding this module, although this concern of being highly interface-based was taken into consideration, when implementing, we could not strictly follow this line of thought. This happened because we are dependent on a tool called solidity2chaincode [37]. However, the output programming language of the translation is *Javascript*, most of the code needed to be adapted in other to meet the specifications of the *Hyperledger Fabric Typescript chaincode*.

Regarding the extension to new smart contract translation, *Osprey* uses the adapter pattern [38], this pattern leverages the incompatibility that each smart contract has between each other. Through the interface *ITranslatorService* users can have the possibility to extend a new smart contract programming language without interfering with the implementations of other translations. This interface offers two functions to be implemented, *translate* and *write* functions. The first function is where it should be the logic about the interpretation of the AST. This function, besides taking the AST as parameter, can take a blockchain connector client. This blockchain connector client offers functions to interact directly with the blockchain, translating behaving as an inline translation. The *write* function is where all the logic about the writing to files should be.

As you can see in Figure 4.2, the translation of a smart contract project is made in three phases. The first phase is when a smart contract project (source and test files) is inputted into the tool. The second phase is where the files are processed. This phase is responsible to read the project files and convert the information within those files into an *Abstract Syntax Tree* (AST). The AST is a way of representing an intermediate language of the information held in the files. After being converted into an AST, that tree is passed to the adapter to be interpreted and translated to the output programming language. This is the last phase.

The translation process works as follows, first it iterates over the AST to translate each dependency of the main smart contract class. The dependencies are expressed in the imports within the file. This process is done to guarantee that in the smart contract where the dependency is being used, the functions and variables are called correctly. Through each dependency found, the tool will search for the path given in the import and translate that file.

When translating a file, the behavior of the tool is, for every class found (classes in *Solidity* are expressed with the contract keyword), it will search first the global variables, then structs, enums, events, modifiers, functions, and object dependencies (in *Solidity* object dependencies are declared using the using keyword). Although in *Hyperledger Fabric chaincodes* there are no global variables, because the state is managed by key-value pair storage, this process is very important to be sure which variables are to be stored as key-value pair when instantiating the *chaincode*. Translating from *Solidity* to *Typescript*, structs are classes which are a representation of many variable types in memory. Enums are datatypes that enable setting predefined constants. Events are variable objects used to signal users when some conditions happen. Modifiers are functions applied to other functions. They are used to specify preconditions to enable or not the execution of the called function. To handle the translation of these types, we have data model classes such as the *EnumBuilder* responsible to translate enum types, *Mixins* responsible to translate the inheritance of classes. *Typescript* does not support multiple inheritances, so a workaround is to use mixins. Mixins are functions that return other functions. *StructBuilder* that are responsible to translate structs in *Solidity* to *Typescript* classes. The *ClassBuilder* class is the main class responsible to combine all data model classes into a single class file. After all, is translated, the adapter wraps all the translations and starts writing them in the proper files. Figure 4.4 shows an overview of this module.

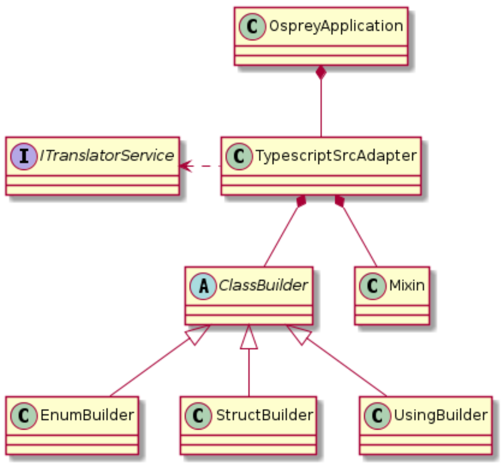


Figure 4.4: Smart Contract Module Overview

4.4.2 Features

Our tool migrates smart contracts written in *Solidity* to *Hyperledger Fabric chaincode* written in *Typescript*. To ensure a perfect migration between those blockchains, the tool should ensure all features that *Solidity* offers, the translated smart contract also offers. Table 4.1 shows the features *Solidity Parser* [20] have and what *Osprey* offers.

Table 4.1 shows us a comparison of what *Solidity Parser* tool [20], with what *Osprey* currently supports. Note that table 4.1 is mostly focused on the current migration in study between smart contracts written in *Solidity* to *chaincodes* written in *Typescript*. Analysing Table 4.1, *Osprey* comparing with *Solidity Parser*, supports almost every feature but those who are specific of the *Ethereum* blockchain. Those specific features are Payables, EVM objects such as the msg object that goes with every transaction triggered over a smart contract, EVM functions such as the function transfer which performs a transfer of assets between two wallets, the balance function responsible to return the amount of cryptocurrency a specific wallet has. The Data Structures such as Mappings and Array and, multiple inheritances are not specific features of the blockchain.

Features	Solidity Parser	Osprey
Modifiers	support	support
Structs	support	support
Events	support	support
Payables	support	not support
Libraries	support	support
Using	support	support
Data Structures (Mappings, Arrays)	support	not support (Future Work)
EVM objects (msg, tx, etc)	support	not support (Future Work)
EVM functions (transfer, send, balance, etc)	support	not support (Future Work)
Imports	support	support
Multiple Inheritance	support	not support (Future Work)

Table 4.1: Comparison between features to migrate *Solidity* smart contracts to *Hyperledger Fabric chaincode* presented by a perfect migration tool and *Osprey*

4.5 Test Module

In this section, we will explain in detail the Test Module, its architecture, how it behaves, and the features it supports in its translation.

4.5.1 Architecture

To guarantee the successful translation of a smart contract on both sides, the original smart contract and translated smart contract, the unit tests of both smart contracts must behave in the same way. This behaves must be coherent because the results expected will be approximated from what the developer tried to test in the test file of the source smart contract. Thus, conclude whether or not the behavior was preserved during the translation process.

Regarding the test translation flow, the *Osprey* test module behaves similar to the smart contract module. First, it will search in the test directory specified as input for the test file which was inputted too. After that, it will transform the information of the source test file into a *Abstract Syntax Tree* (AST). This process is made using a package called *accorn*. *Accorn* is a package that converts *Javascript* files into *Abstract Syntax Tree*. Once the AST is built, Osprey iterates over it and, node by node it translates to *Typescript*. After the translation is done, Osprey produces a test file to be used in *Hyperledger Fabric*. Also, integrating Osprey in *Hyperledger Cactus*, our tool can have two major features: (i) provide an end to end translation and, (ii) run, automatically, those tests translated. *Cactus*, could be seen as a framework-as-a-service, where it provides mechanisms to get the source smart contracts and, to instantiate two ledgers, one to run the source smart contract, and the other to run the translated smart contract.

In terms of architecture, the *Osprey* test module was designed the same way as the smart contract module, using the adapter design pattern. This pattern allowed us to have the same feature that the smart contract module has, flexibility. Flexibility, because we can have multiple output translation implementations without compromising the entire structure of the tool. Also, in this module, we used another design pattern to complement the adapter. This pattern is called Factory design pattern [39]. It allows us to decide which instance of the test translation to use based on the input of the user. Based on that input, Osprey can use an instance to translate the test files and output the files in the chosen programming language.

In Figure 4.5 we can see the test module architecture.

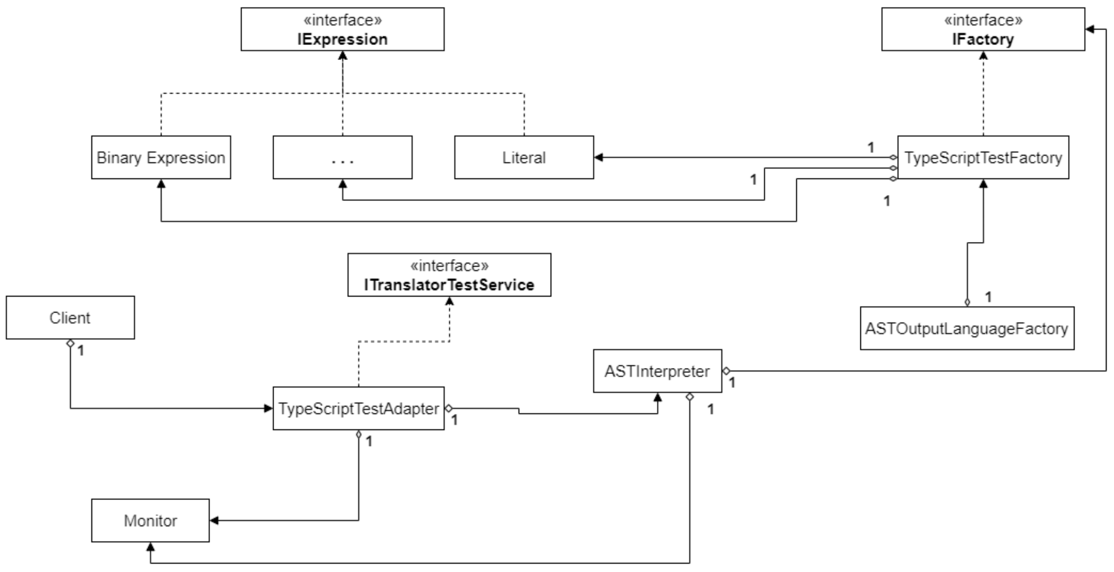
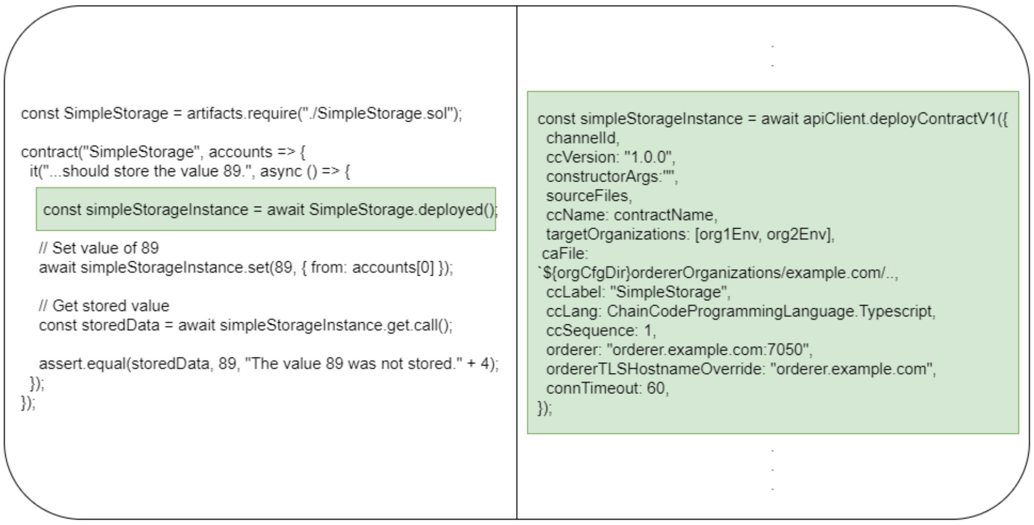


Figure 4.5: Test Module architecture overview

As presented the architecture in its genesis, is not different from the one we saw in Figure 4.4 from the Smart Contract Module. However, we can see a new component added to the architecture in both modules, Test and Smart Contract modules, called Monitor. This decision was made, because the fact we needed to not only track where the smart contract translations were outputted, but also to track in the tests' translation process when we were facing smart contract function calls or calls to other packages used in the translation. It is the monitor's job to track that kind of information as long as the iteration and translation process occurs. For instance, in Figure 4.6 we can observe on the left a call to a smart contract function. The monitor will save that information, and when the translation occurs we can see on the right side that it was adapted to a smart contract function call used on the *Cactus* test file template.

Furthermore, a decision in the test translation process was made. We didn't include the assertions packages used in the source test file (i.g. *chai*, *bigint*, among others). This decision was made because, in the Cactus template, those packages are being used. Also, we ignore the first test function in the



```
const SimpleStorage = artifacts.require("./SimpleStorage.sol");

contract("SimpleStorage", accounts => {
  it("...should store the value 89.", async () => {

    const simpleStorageInstance = await SimpleStorage.deployed();

    // Set value of 89
    await simpleStorageInstance.set(89, { from: accounts[0] });

    // Get stored value
    const storedData = await simpleStorageInstance.get.call();

    assert.equal(storedData, 89, "The value 89 was not stored." + 4);
  });
});
```

```
.
.
const simpleStorageInstance = await apiClient.deployContractV1({
  channelId,
  ccVersion: "1.0.0",
  constructorArgs:"",
  sourceFiles,
  ccName: contractName,
  targetOrganizations: [org1Env, org2Env],
 caFile:
`${orgCfgDir}ordererOrganizations/example.com/..,
  ccLabel: "SimpleStorage",
  ccLang: ChainCodeProgrammingLanguage.Typescript,
  ccSequence: 1,
  orderer: "orderer.example.com:7050",
  ordererTLSHostnameOverride: "orderer.example.com",
  connTimeout: 60,
});
.
.
.
```

Figure 4.6: Monitor job overview

source test file, the one with the keyword contract. This decision was made, based on the fact that the test translation will be wrapped up in a test function from the Cactus template.

4.6 Implementation

Osprey tool was implemented not only as a standalone migrator tool, but also as a plugin integrated into *Hyperledger Cactus*. *Osprey* was developed in *Typescript*, to use the interoperability between blockchains that *Cactus* framework offers, such as *Hyperledger Fabric* connector, *Hyperledger Besu* connector.

Mainly *Hyperledger Cactus* complements Osprey such that it can successfully perform smart contract migration between the various blockchains. As stated before, this migration process takes at least 4 connections to both blockchains in the migration process, the source, and the target blockchains. These four connections provided by *Cactus* are made through connectors. Connectors are implementations that allow blockchains to have interoperability between them, as well as guaranteeing all the security specifications that each transaction must have in their respective blockchain. Also, these connections are respectively to obtain the smart contract to be translated and then migrated; Then to instantiate two test ledgers of both blockchains, the source and the target blockchains; And, after that to deploy the smart contract in the target blockchain.

Looking at the connections that Osprey must have to complete a full smart contract migration, it states the category where we consider *Hyperledger Cactus* as a blockchain-as-a-service. Blockchain-as-a-service, since it allows *Osprey* connect to both blockchains to perform transactions over them and, also to build a test infrastructure to test the smart contract, before deploying them in the real target blockchain environment.

5 Evaluation

Contents

5.1 Osprey Evaluation

In this section, we evaluate the translation of heterogeneous smart contracts using our tool *Osprey*. The metrics used on the evaluation process were the time elapsed on every translation of each smart contract and, in the end we discuss aspects about the tool's utility.

5.2 Setup and Test Environment

All tests were made using a 16 GB RAM machine with a AMD Ryzen 5 3600 CPU and 480 GB SSD of storage. We have put a dataset of 13 *Solidity* smart contracts to test. The test was the translation of each smart contract of the dataset and measure the time that *Osprey* took to translate them. The number of times that the dataset was translated was 10000 times.

5.3 Translation Evaluation

We gathered a total of 13 smart contracts in *Solidity* and execute performance tests to validate the time elapsed of translating each smart contract from the dataset. To ensure that each translation was successfully, we manually test the output *Osprey* gave. The smart contracts used were collected mostly through *Github* and from the SmartBugs dataset [2]. Figure 5.1 shows the time elapsed of each translation and the average of all smart contracts translated after 10000 times.

Smart Contract	Time Elapsed
AdvanceStorage.sol	2.58 ms
Greeter.sol	1.82 ms
IntegerOverflowAdd.sol	1.86 ms
IntegerOverflowBenign1.sol	1.96 ms
IntegerOverflowMinimal.sol	2.20 ms
IntegerOverflowMul.sol	3.25 ms
IntegerOverflowMultiTxMultiFuncFeasible.sol	4.20 ms
IntegerOverflowMultiTxOneFuncFeasible.sol	3.94 ms
IntegerOverflowSingleTransaction.sol	3.35 ms
Overflow.sol	4.17 ms
Overflow_Add.sol	2.65 ms
QueueMapping.sol	7.59 ms
SimpleStorage.sol	4.42 ms
	Average = 3.68 ms

Table 5.1: *Osprey Solidity* smart contracts dataset translation time test, translating 10000 times the entire dataset

Analysing table 5.1 we can verify that the faster smart contract translation was the *Greeter.sol*, a simple smart contract that has one constructor and one function. Contrarily we can verify that the slowest smart contract translation was the *QueueMapping.sol*, a more complex smart contract where it envolves structures and other more complex operations. Although the throughput and latency were not tested, we can infer this results are acceptable.

5.4 Readability

In this section we will measure the readability of the smart contract translation outputted by *Osprey*. Since the tool will be used to perform smart contract migration between heterogeneous blockchains, whether to make a full end-to-end migration or to translate the smart contract to a new blockchain, to continue the development process, it is important the smart contract *Osprey* outputs be readable and structured to lower the refactoring code by the developers.

To have some insight on how *Osprey* translated and structured the smart contracts it outputs, we conducted a survey where we tried to reach out the people that were familiar with the blockchain technology and the smart contract programming. In this survey we had a total of 17 participants. We tried to focus mainly on the translation process and how they classify the smart contract regarding the readability and the structure of the produced code. Furthermore, we tried to divide the smart contracts regarding its complexity. As mentioned, our dataset is composed by 13 smart contracts, out of those 13 smart contracts we chose 2. The complexity of the smart contracts chosen, was measure by the amount of functions, global variables and the extensibility of the contract, in terms of the number of lines of code, each function presented. Based on this premise we classify the smart contracts presented in Appendix A, as simple and complex respectively (A.1, A.2).

Once *Osprey*, as mentioned before, can be used as a standalone migration tool, in order to users use our tool to migrate smart contracts, and continue the development process, it is important to evaluate how well the translated smart contract is readable and organized. For this purpose, we call the readability of the tool. This is justified, because in software development industry, code readability is a key aspect for the maintenance product and project. Also, if the code is organized and well structured, it is easier for developers to add more features and maintain the project/product.

For the answers to the survey be more accurate, we tried to minimize the scope of people that answered. That scope tried to involved people that were familiarized with blockchain technology, blockchain product delivery and, smart contract development, both in *Solidity* and *Hyperledger Fabric typescript chaincode* development. Figure 5.1 shows the scope of people who answered the survey.

From Figure 5.1, leading the percentage of the answers was people whose job title is related to Academic environment, being professors, students, researchers with 71.4% of the answers. Following the lead were the Software Developers with 35.5%, and at last were the product owners with 21.4% of the answers. From this 14 people who answer the survey the greatest part of were not familiarized with both *Solidity* and *Chaincode* programming. Figure 5.2 show the percentage of each column, being 1 not

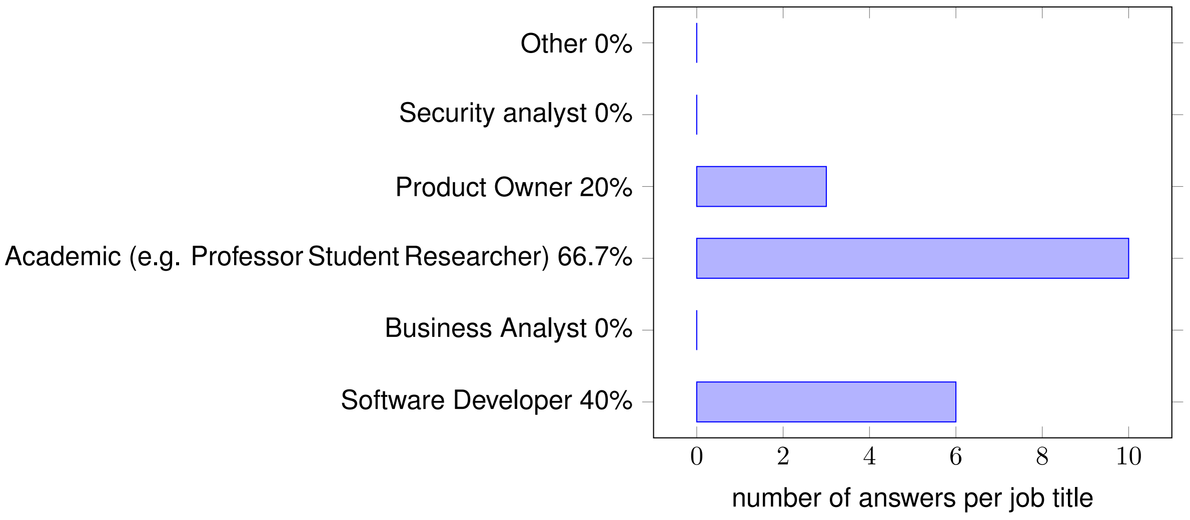


Figure 5.1: Job title of the users who answer the survey

familiarized with the programming language, and 5 being familiarized with the programming language.

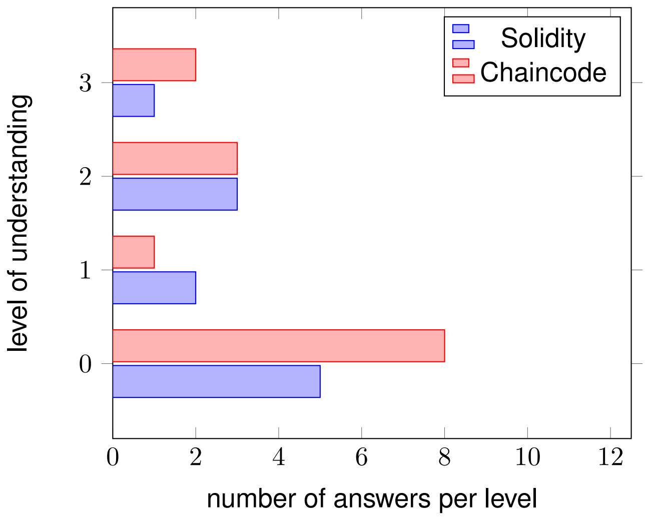


Figure 5.2: Level of understanding on *Solidity* and *Chaincode* programming

Moreover, from the total of people who answer this survey, most of them only have at least one year of experienced with blockchain technology. On the other hand, three out of fourteen have three or more years of experience with blockchain technology. Figure 5.3 show the statistics.

Regarding the translation of smart contracts, the translation of the simple smart contract Appendix A [A.1] show very impressive results. Most answers were on the level five (levels one to five, one being do not understand and, five understand) on understanding the smart contract and their translation. On the other hand, the results about the structure of the translation, although, most of it is on well structured (level four), it shows that the tool itself has a lot of improvements to be made, in order to become a very

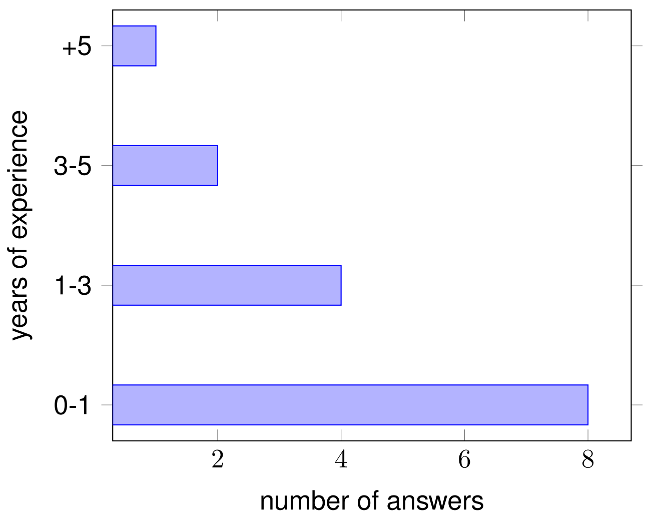


Figure 5.3: Years of experience with blockchain technology

good migration tool. Figure 5.4 shows the results.

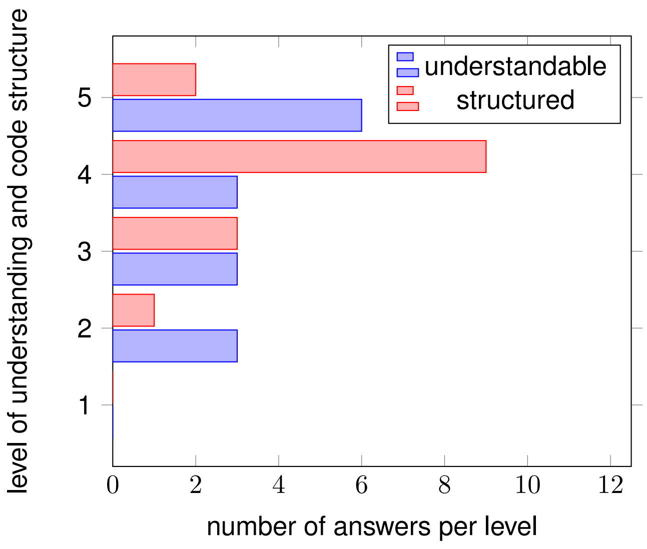


Figure 5.4: Number of answers regarding the evaluation of the understanding and the structure of the translated code on a simple smart contract A.1

Going deeper in the complexity of the translation, we present a more complex *Solidity* smart contract (Appendix A [A.2]) and their respective translation. In this scenario, Figure 5.5 shows that the tool itself have a lot space to improve. This is because, most of the answers were divided between the level two and four. Level two means that they didn't understand very much the smart contract and, its translation. However, most of the answers classified the translation as being fairly good structured.

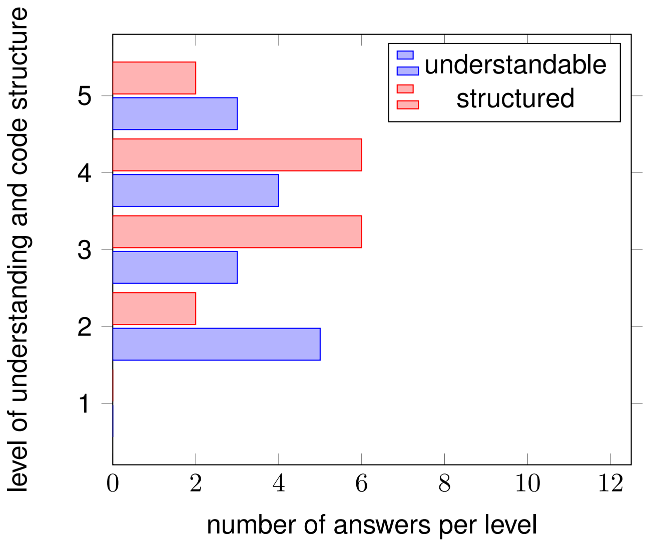


Figure 5.5: Number of answers regarding the evaluation of the understanding and the structure of the translated code on a complex smart contract A.2

6 Conclusion

Contents

This research presents you Osprey, a smart contract migrator tool between heterogeneous blockchains. Osprey helps blockchain interoperability take a step further on making companies use blockchain technology without being afraid of the costs of maintenance or afraid to start over when a blockchain becomes obsolete or even when they find another blockchain that offers more appealing features. Integrated in *Hyperledger Cactus*, a blockchain connector, as a plugin, Osprey can (i) establish a connection with *Hyperledger Besu* and *Hyperledger Fabric*; (ii) acquire the smart contracts from *Ethereum* blockchain through *Hyperledger Besu*; (iii) use a blockchain validator, before and after deliver the contracts to the migrator, to evaluate/analyse them; and (iv) issue the transactions in *Hyperledger Fabric* blockchain. Also, another reason on integrating our tool in *Cactus*, the ability to instantiate and destroy blockchain instances to test the functionality of the smart contracts. After the original and translated smart contracts are tested, and the ledgers destroyed, both execution outputs are compared to validate their correctness and behavior. Moreover, Osprey can be used as a blockchain cloud migration tool solution where it can be deployed in the cloud and used by the community. The experimental results over a dataset with 13 *Solidity* smart contracts, shows that Osprey in average can perform translations in about 3.68 milliseconds. Although this is a solution that can be extended and can be improved in more features, this is a contribution to help blockchain community move forward on blockchain smart contract migration and help future works on blockchain interoperability solutions.

6.0.1 Contributions

This research allows blockchain technology to take a step further and contribute to the interoperability of blockchains, by allowing the migration of smart contracts between heterogeneous blockchains. At the time this study is made, the solutions available are only theoretical ones, such as Hyperservice [11]. Also, the only solution found at this time was the solidity parser solution [12]. A solution that is a proof of concept that blockchain migration between heterogeneous blockchains can be made. However, it is an old solution that has not been maintained nor updated and, it only performs migrations between *Solidity* smart contracts to *Javascript* chaincode. This means it lacks flexibility.

Osprey on the other hand is a tool flexible, which first starts to migrate *Solidity* smart contracts to *Typescript* chaincode, but it can be extended to perform migrations between other types of smart contracts. Our contributions are as follows.

1. Translate smart contracts written in *Solidity* to *Typescript*.
2. Design a tool, able to be flexible, meaning it can be extended to migrate smart contracts from other blockchains and whose smart contract programming languages are different from *Solidity* as input and *Typescript* as output.
3. Develop translation mechanisms for unit tests between both blockchains, source, and target.

4. Contribute to *Hyperledger Cactus* becoming a more complete tool in terms of blockchain interoperability.

6.1 Future Work

This work help the subject of smart contract migration on a new level, being one of the tools implemented with almost full translation of *Ethereum* smart contracts written in *Solidity*. *Osprey* helps companies to not being afraid of migrating their projects when it's development reaches a critical stage (e.g. the maintenance cost of the project is too high, or some vulnerabilities were discover in the current blockchain their working on). Furthermore, to turn each individual module of the tool more complete, in the future we plan to tackle the features, in the smart contract module 4.4, that were not implemented. After that we, plan to disconnect completely from the dependency held on the solidity2chaincode tool [37], turning the implementation of this module, similar to the test module.

Regarding the test module (Section 4.5), for future work we plan to move further on the implementation of the features that are not implemented and, to implement a way of having inline test calls. The inline testing feature, allow at runtime, when the test ledgers are instantiated the code can automatically be called over those ledgers, becoming the migration process and testing more automatic.

Bibliography

[1] E. Androulaki, A. Barger, V. Bortnikov, S. Muralidharan, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Murthy, C. Ferris, G. Laventman, Y. Manevich, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains,” in *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, vol. 2018-January. Association for Computing Machinery, Inc, apr 2018.

[2] “smartbugs/dataset at master · smartbugs/smartbugs.” [Online]. Available: https://github.com/smartbugs/smartbugs/tree/master/dataset

[3] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, “A Survey on Blockchain Interoperability: Past, Present, and Future Trends,” 2020. [Online]. Available: http://arxiv.org/abs/2005.14282

[4] R. Belchior, M. Correia, and A. Vasconcelos, “Justicechain: Using blockchain to protect justice logs,” in *OTM Confederated International Conferences” On the Move to Meaningful Internet Systems”*. Springer, 2019, pp. 318–325.

[5] M. Mettler, “Blockchain technology in healthcare: The revolution starts here,” in *2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom)*, 2016, pp. 1–3.

[6] M. Turkanović, M. Hölbl, K. Košič, M. Heričko, and A. Kamišalić, “Eductx: A blockchain-based higher education credit platform,” *IEEE Access*, vol. 6, pp. 5112–5127, 2018.

[7] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang, “Fine grained, secure and efficient data provenance on blockchain systems,” *Proceedings of the VLDB Endowment*, vol. 12, no. 9, pp. 975–988, 2018.

[8] H. D. Bandara, X. Xu, and I. Weber, “Patterns for Blockchain Migration,” pp. 1–40, 2019. [Online]. Available: http://arxiv.org/abs/1906.00239

[9] "Ethereum Whitepaper — ethereum.org." [Online]. Available: https://ethereum.org/en/whitepaper/{#}ethereum

[10] "cactus/whitepaper.md at master · hyperledger/cactus." [Online]. Available: https://github.com/hyperledger/cactus/blob/master/whitepaper/whitepaper.md

[11] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, and Y. C. Hu, "Hyperservice: Interoperability and programmability across heterogeneous blockchains," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 549–566, 2019.

[12] M. A. Zafar, F. Sher, M. U. Janjua, and S. Baset, "SOL2JS: Translating solidity contracts into Javascript for hyperledger fabric," *SERIAL 2018 - Proceedings of the 2018 Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pp. 19–24, 2018.

[13] "Hyperledger Besu Enterprise Ethereum Client - Hyperledger Besu." [Online]. Available: https://besu.hyperledger.org/en/stable/

[14] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A Survey on Blockchain Interoperability: Past, Present, and Future Trends," *ACM Computing Surveys*, vol. 54, no. 8, pp. 1–41, may 2021. [Online]. Available: http://arxiv.org/abs/2005.14282

[15] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," Tech. Rep. [Online]. Available: www.bitcoin.org

[16] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Čapkun, "On the security and performance of Proof of Work blockchains," in *Proceedings of the ACM Conference on Computer and Communications Security*, vol. 24-28-October-2016. Association for Computing Machinery, oct 2016, pp. 3–16.

[17] "Solidity — Solidity 0.8.0 documentation." [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/

[18] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 530–541.

[19] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: a framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1349–1352.

[20] "tool to migrate solidity do javascript." [Online]. Available: https://github.com/hyperledger-labs/solidity2chaincode

[21] "Ieee xplore full-text pdf:." [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9284781&casa_token=VPiHj4dwcqsAAAAA:Wg-id3QVLRu5xA0YiferiOHvJXcPHnALB_J0Fkgig9yX0f-hui8fIYMeVhM4M7blAs37zYLrLg&tag=1

[22] P. Frauenthaler, M. Sigwart, C. Spanring, and S. Schulte, "Testimonium: A cost-efficient blockchain relay," 2 2020. [Online]. Available: http://arxiv.org/abs/2002.12837

[23] "ethereum/btcrelay: Ethereum contract for bitcoin spv: Live on https://etherscan.io/address/0x41f274c0023f83391de4e0733c609df5a124c3d4." [Online]. Available: https://github.com/ethereum/btcrelay

[24] J. F. Snyder, M. A. Ratner, H. Wang, D. He, X. Wang, C. Xu, W. Qiu, Y. Yao, and Q. Wang, "An electricity cross-chain platform based on sidechain relay you may also like on the sensitivity of protein data bank normal mode analysis: an application to gh10 xylanases monique m tirion-ion conductivity of comb polysiloxane polyelectrolytes containing oligoether and perfluoroether sidechains an electricity cross-chain platform based on sidechain relay," *Journal of Physics: Conference Series*, vol. 1631, p. 12189, 2020.

[25] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," 2017. [Online]. Available: https://plasma.io/

[26] H. Tian, K. Xue, S. Li, J. Xu, J. Liu, and J. Zhao, "Enabling cross-chain transactions: A decentralized cryptocurrency exchange protocol."

[27] W. Warren and A. Bandeali, "0x: An open protocol for decentralized exchange on the ethereum blockchain," 2017.

[28] "Dextt: Deterministic cross-blockchain token transfers — enhanced reader."

[29] J. Burdges, A. Cevallos, P. Czaban, R. Habermeier, S. Hosseini, F. Lama, H. K. Alper, X. Luo, F. Shirazi, A. Stewart, and G. Wood, "Overview of polkadot and its design considerations."

[30] "Ieee xplore full-text pdf:." [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8431965&casa_token=O5XrC_13FWwAAAAA:hIlu7JDIBPlwOWDRW9VDXbRcpb27kgR--akUa5pmJbc2snkoizqbSHDdVFO6xRY7aAf75-6CJg

[31] "Ieee xplore full-text pdf:." [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8743548&casa_token=kiFdvnVki4MAAAAA:U0stBTODdpT8NsPOpUh19Ri4GvKhbiEfmUp6EfWoKHQXCYolJLYeSgqrSA6gUCabGHr6GaQYvA

[32] X. Wang, T. Tawose, F. Yan, and D. Zhao, "Distributed nonblocking commit protocols for many-party cross-blockchain transactions."

[33] R. Belchior, A. Vasconcelos, M. Correia, and T. Hardjono, “Enabling Cross-Jurisdiction Digital Asset Transfer,” in *IEEE International Conference on Services Computing*. IEEE, 2021.

[34] H. Montgomery, H. Borne-Pons, J. Hamilton, M. Bowman, P. Somogyvari, S. Fujimoto, R. Belchior, T. Kuhrt, and T. Takeuchi, “cactus/whitepaper.md at master · hyperledger/cactus.” [Online]. Available: https://github.com/hyperledger/cactus/blob/master/whitepaper/whitepaper.md

[35] “the-super-tiny-compiler/the-super-tiny-compiler.js at master · jamiebuilds/the-super-tiny-compiler.” [Online]. Available: https://github.com/jamiebuilds/the-super-tiny-compiler/blob/master/the-super-tiny-compiler.js

[36] “theliso/cactus: Hyperledger cactus is a new approach to the blockchain interoperability problem.” [Online]. Available: https://github.com/theliso/cactus/tree/main/packages/cactus-plugin-blockchain-migrator

[37] “hyperledger-labs-archives/solidity2chaincode: This tool converts solidity contract into javascript chaincode through source-to-source translation for running them onto hyperledger fabric.” [Online]. Available: https://github.com/hyperledger-labs-archives/solidity2chaincode

[38] “Adapter.” [Online]. Available: https://refactoring.guru/design-patterns/adapter

[39] “Best Practice Software Engineering - Factory Method.” [Online]. Available: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/factory.html

A

Code of Project

Listing A.1: Simple Solidity Smart Contract used in the form

```
pragma solidity ^0.8.7;

contract SimpleStorage {
    uint256 storedData;

    function set(uint256 x) public {
        storedData = x;
    }

    function get() public view returns (uint256) {
        return storedData;
    }
}
```

Listing A.2: Simple Solidity Smart Contract used in the form

```
pragma solidity ^0.5.16;

contract QueueMapping {
    uint256 private front;
    uint256 private back;
    mapping(uint256 => uint256) private queue;

    constructor() public {
        front = 1;
        back = 0;
    }
    ...
    function enqueue(uint256 _data) public {
        require(back + 1 > back, "The queue is full.");
        //Increment back and set data
        back++;
        queue[back] = _data;
    }
    ...
}
```

Listing A.3: Greeter Solidity smart contract

```
pragma solidity 0.8.7;

contract Greeter {
    string greeting;

    constructor(string memory _greeting) {
        greeting = _greeting;
    }

    function greet() public view returns (string memory) {
        return greeting;
    }
}
```

Listing A.4: AST generated from Listing A.3

```
{
    "type": "SourceUnit",
    "children": [
        {
            "type": "PragmaDirective",
            "name": "solidity",
            "value": "0.8.7"
        },
        {
            "type": "ContractDefinition",
            "name": "Greeter",
            "baseContracts": [],
            "subNodes": [
                {
                    "type": "StateVariableDeclaration",
                    "variables": [
                        {
                            "type": "VariableDeclaration",
                            "typeName": {
                                "type": "ElementaryTypeName",
                                "name": "string"
                            },
                            "name": "greeting",
                            "expression": null,
                            "visibility": "default",
                            "isStateVar": true,
                            "isDeclaredConst": false,
                            "isIndexed": false
                        }
                    ],
                    "initialValue": null
                },
                {
                    "type": "FunctionDefinition",
                    "name": null,
                    "parameters": {
                        "type": "ParameterList",
                    "parameters": [
                        {
                            "type": "Parameter",
                            "typeName": {
                                "type": "ElementaryTypeName",
                                "name": "string"
                            },
                            "name": "_greeting",
                            "storageLocation": "memory",
                            "isStateVar": false,
                            "isIndexed": false
                        }
                    ]
                },
                "body": {
                    "type": "Block",
                    "statements": [
                        {
                            "type": "ExpressionStatement",
                            "expression": {
                                "type": "BinaryOperation",
                                "operator": "=",
                                "left": {
                                    "type": "Identifier",
                                    "name": "greeting"
                                },
                                "right": {
                                    "type": "Identifier",
                                    "name": "_greeting"
                                }
                            }
                        }
                    ]
                },
                "visibility": "default",
                "modifiers": [],
                "isConstructor": true,
                "stateMutability": null
                },
                {
                    "type": "FunctionDefinition",
                    "name": "greet",
                    "parameters": {
                        "type": "ParameterList",
                        "parameters": []
                    },
                    "returnParameters": {
                        "type": "ParameterList",
                        "parameters": [
                            {
                                "type": "Parameter",
                                "typeName": {
                                    "type": "ElementaryTypeName",
                                    "name": "string"
                                },
                                "name": null,
                                "storageLocation": "memory",
                                "isStateVar": false,
                                "isIndexed": false
                            }
                        ]
                    },
                    "body": {
                        "type": "Block",
                        "statements": [
                            {
                                "type": "ReturnStatement",
                                "expression": {
                                    "type": "Identifier",
                                    "name": "greeting"
                                }
                            }
                        ]
                    },
                    "visibility": "public",
                    "modifiers": [],
                    "isConstructor": false,
                    "stateMutability": "view"
                }
            ],
            "kind": "contract"
        }
    ]
}
```