OSPREY: Migrating Smart Contracts Between Heterogeneous Blockchains

Luís Abrunhosa

***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.**

I. INTRODUCTION

Blockchain technology has grown over the decade [10] and drawn the attention in several areas: audits [9], health care [12], education [14], among others. It is a decentralized system, offering privacy, security, transparency, and data immutability [13].

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[10]. 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, that allows to establish a connection between the pair of blockchains in study, *Ethereum* blockchain [4] and *Hyperledger Fabric* [13]. 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 [3], 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 [15] that extracts information from the *Solidity* smart contracts, and through a converter, it converts that information into *Hyperledger Fabric* chaincode.

A. 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.

II. RELATED WORK

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 [15] 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.

III. 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].

A. 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.

B. 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.

IV. 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 [15], was developed with the gold 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)".

A. 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.

B. 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 [15] (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*.

V. OSPREY OVERVIEW

To explore the possibility of migrating smart contracts between heterogeneous blockchains, we design *Osprey* [6]. 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*.

VI. 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*).

VII. 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*.

As Figure 1 shows, the first part of the process of migrating smart contracts between heterogeneous blockchains is the

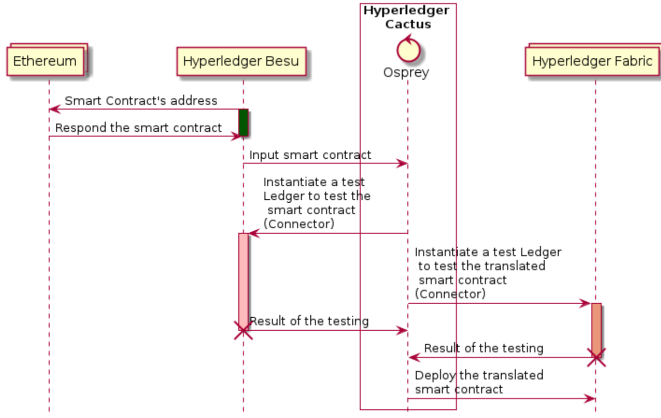

Fig. 1: Sequence diagram of the translation process integrated with *Hyperledger Cactus*

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 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*.

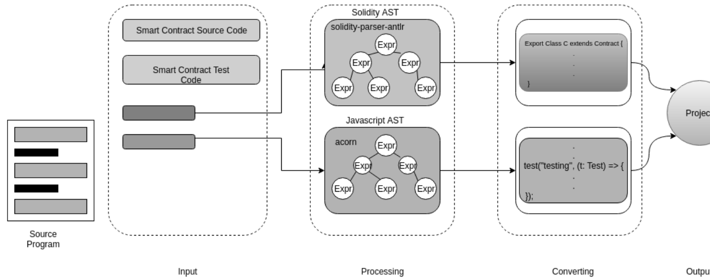

Fig. 2: Osprey flow overview

Regarding the translation flow, Figure 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 VIII, 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 IX, 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 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 3 demonstrates the communication.

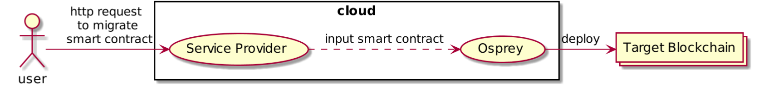

Fig. 3: Osprey as microservice in the cloud

VIII. Smart Contract Module

As mentioned before, *Osprey* architecture is divided into two modules, the Smart Contract Module, and the Test Module described in Section IX. 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.

A. Architecture

Osprey was designed to be highly interface based, especially in the Test Module IX. 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 [5]. 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 [1], 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 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 shows an overview of this module.

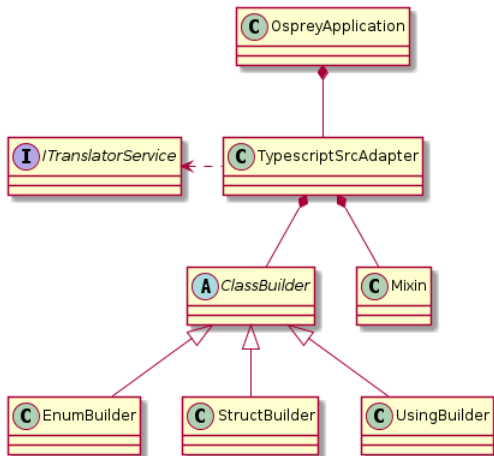

Fig. 4: Smart Contract Module Overview

B. 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 I shows the features *Solidity Parser* [7] have and what *Osprey* offers.

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

Table I shows us a comparison of what *Solidity Parser* tool [7], with what *Osprey* currently supports. Note that table I is mostly focused on the current migration in study between smart contracts written in *Solidity* to *chaincodes* written in *Typescript*. Analysing Table I, *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.

IX. 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.

A. 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 [2]. 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 5 we can see the test module architecture.

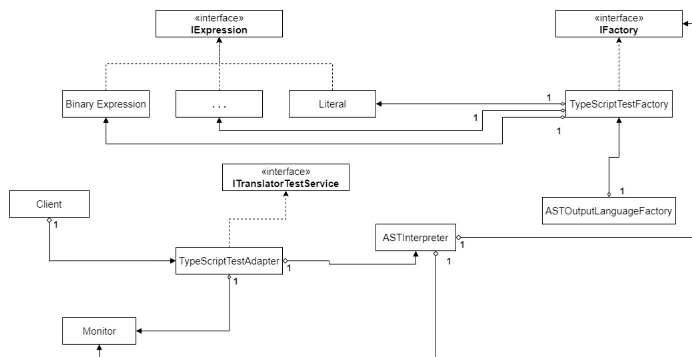

Fig. 5: Test Module architecture overview

As presented the architecture in its genesis, is not different from the one we saw in Figure 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 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 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.

X. 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

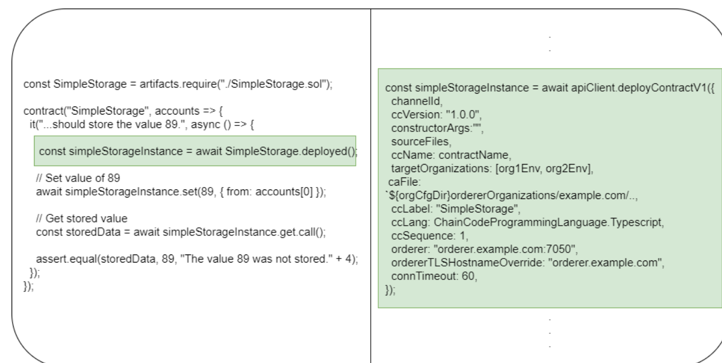

Fig. 6: Monitor job overview

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, because it allows Osprey to 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.

XI. Conclusion

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.

A. 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 [15]. 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) Becoming *Hyperledger Cactus* a more complete tool in terms of blockchain interoperability.

XII. 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 VIII, that were not implemented. After that we, plan to disconnect completely from the dependency held on the solidity2chaincode tool [5], turning the implementation of this module, similar to the test module.

Regarding the test module (Section IX), 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.

References

[1] Adapter.

[2] Best Practice Software Engineering - Factory Method.

[3] cactus/whitepaper.md at master · hyperledger/cactus.

[4] Ethereum Whitepaper — ethereum.org.

[5] hyperledger-labs-archives/solidity2chaincode: This tool converts solidity contract into javascript chaincode through source-to-source translation for running them onto hyperledger fabric.

[6] theliso/cactus: Hyperledger cactus is a new approach to the blockchain interoperability problem.

[7] tool to migrate solidity do javascript.

[8] H. D. Bandara, X. Xu, and I. Weber. Patterns for Blockchain Migration. pages 1–40, 2019.

[9] 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"*, pages 318–325. Springer, 2019.

[10] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia. A Survey on Blockchain Interoperability: Past, Present, and Future Trends. 2020.

[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*, pages 549–566, 2019.

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

[13] 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*, 12(9):975–988, 2018.

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

[15] 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*, pages 19–24, 2018.