Toulouse INP – ENSEEIHT
École nationale supérieure d'électrotechnique, d'électronique, d'informatique, d'hydraulique et des télécommunications

Internship Report
Performed at LIST CEA

29th March 2021 – 26 September 2021

Design an implementation of an agent-based simulation model for the Hyperledger Fabric Blockchain Protocol

Chaïmaa Benabbou
chaimaa.benabbou@etu.toulouse-inp.fr
3rd year – HPC & Big Data field
Master PSMSC
ENSEEIHT

LIST CEA
F-91120, Palaiseau, France

CEA Tutor
Önder Gürcan
onder.gurcan@cea.fr

ENSEEIHT Tutor
Philippe Queinnec
philippe.queinnec@enseeiht.fr

Abstract

Smart contracts are programs stored on a blockchain that run when predetermined conditions are met. Blockchain and smart contract based solutions can facilitate secure exchanges within a critical application. To do this, it is proposed to provide a simulation model to assist in the analysis, development and implementation of safe protocols and smart contracts.

However, designing and implementing a smart contract is not trivial since upon deployment on a blockchain, it is no longer possible to modify it (neither for improving nor for bug fixing). It is only possible by deploying a new version of the smart contract which is costly (deployment cost for the new contract and destruction cost for the old contract). To this end, there are many solutions for testing the smart contracts before their deployment. Since realizing bug-free smart contracts increase the reliability, as well as reduce the cost, testing is an essential activity.

In this paper, we first carry out a State-of-the-art to group the existing solutions that attempt to tackle smart contract verification, validation and testing (VV&T) into following categories: public test networks, security analysis tools, blockchain emulators and blockchain simulators. Then, we analyze these solutions, categorize them and show what their pros and cons are.

Secondly, we focus on the main objective of this internship which is developing an agent-based model to analyse by simulation the operational safety from the angle of the availability of the consensus protocol of the Hyperledger blockchain using an agent-based simulation. To this end, various hypotheses will be tested in order to build a solid design for applications that use Hyperledger Fabric.

Finally, we validate our implementation considering an Energy Performance Contracts (EPC) use case, a prototype to store and validate building energy consumption based on [18].

1 Acknowledgment

Many thanks to my adviser, GÜRCAN Önder, who offered me patience, guidance, support and encouragement with a perfect blend of insight and helped make some sense of the confusion throughout this project. His expertise was invaluable in formulating the research questions and methodology. I have benefited greatly from his wealth of knowledge and meticulous editing. Also, thanks to the laboratory members, the head of the laboratory TUCCI Sara, DUBAILLAY Pierre, ROUSILLE Hector who read my numerous revisions and checked my work, not forgetting the other members who provided support and a number of helpful comments and suggestions. Your encouraging words and thoughtful, detailed feedback have been very important to me. I'm proud of, and grateful for, my time working in the LICIA (CEA LIST).

I would also like to thank my ENSEEIHT teachers for their valuable guidance throughout my studies. You provided me with the tools that I needed to choose the right direction and successfully complete my engineering and master degree.

Lastly, my family deserves endless gratitude. Thank you to my parents, Benabbou EL Arbi and Amer Khadija for your endless support. You have always stood behind me, and this was no exception. Thank you to my sisters for always being there for me and giving me enough moral support, encouragement and motivation to accomplish my goals.

To my family, I give everything, including this.

Contents

2 Introduction

With Bitcoin, the blockchain technology initially gained traction in 2008 [35]. Nowadays, it is considered as a major disruptive innovation with the potential to transform most industries. The blockchain can be regarded as a transparent, secure technology for storing and transmitting information that operates without a trusted third party.

For making the blockchain a general-purpose solution, Ethereum introduced the concept of smart contracts [43], *i.e.* immutable code on the blockchain that gets executed automatically once certain conditions are met between parties that do not necessarily trust each other. With smart contracts, the blockchain technology have seen a rapid climb to prominence, with its applications in various domains. They can improve insurance processes by automating claims when certain events occur, enable better supply chains[1] and more applications in business, commerce, and governance are still emerging . Given the wide range of smart contract adoption, best practices for implementing such code must be taken.

Since smart contract deployment is definitive, the vulnerabilities and the attacks can challenge the sustainability of applications using them. The most known ones are related to Ethereum: the DAO attack in May 2016[2] that helped to gather around $150 million or the Parity Wallet hack[3], on November 8, 2017, that valued at over $152 million. Moreover, executing smart contracts consumes certain amount of computation and memory. For instance, Ethereum community introduced *gas*, a notion to measure the amount of computational power needed to be paid to the miners in order to execute their operations.

Today, there are several blockchain systems that support smart contracts, such as Ethereum [43] and Qorum [6] that use Solidity for writing smart contracts, Hyperledger Fabric [3] that utilizes general-purpose programming languages (Go, Node.js, JavaScript and Java), Corda [9] that supports Java and Kotlin languages, Stellar [29] that uses Javascript SDK and Node.js and NEO [28] that develops smart contracts using C#. Besides, Tezos[17] designed new functional programming languages for smart contracts: Michelson, SmartPy and LIGO. Recently, Cardano[4] introduced smart contracts using Plutus, Marlowe & Glow programming languages. Besides, Tendermint/Cosmos[5] supports smart contracts written in any languages. Bitcoin is also a pioneer in this field by enabling writing transactional rules using its non Turing-complete language, called Script[6] and soon a completely new way of designing smart contracts in Bitcoin will be introduced.

A smart contract is a set of immutable code which automatically executes, verifies and facilitates changes to state objects in transactions and control actions defined. In that way, understanding how smart contracts work is essential for the rest of our project.

The life-cycle of a smart contract consists of the following stages [18, 20, 27, 40]:

1. Analysis and design of the intended smart contract:
    - Define the requirements and the use-cases of the smart contract (*i.e.* the functions, the target blockchain, the target language, etc.).
    - Design the clauses and the functions of the smart contract.
2. Implementation and testing of the smart contract:

[1] Smart Contract and Supply Chain, `https://www.frontiersin.org/articles/10.3389/fbloc.2020.535787/full`, last access on 28/06/2021.

[2] DAO attack cost, `https://blog.b9lab.com/the-dao-hack-in-eight-minutes-94919018692d`, last access on 07/06/2021.

[3] Parity Wallet hack cost, `https://cointelegraph.com/news/parity-multisig-wallet-hacked-or-how-come`, last access on 07/06/2021.

[4] Cardano, `https://docs.cardano.org/en/latest/index.html`, last access on 16/06/2021.

[5] Tendermint, `https://tendermint.com/core/`, last access on 18/06/2021.

[6] Script, `https://en.bitcoin.it/wiki/Script`, last access on 15/06/2021.

- Select a testing approach and implement the smart contract using a corresponding language.
- Perform tests and analyze the results.

3. Repetition of the above steps till the desired smart contract is implemented in the target language for the target blockchain.
4. Deployment of the smart contract onto blockchain network definitively.
5. Execution of the smart contract through functions calls:
   - Select a smart contract function and send a function call in the form of transaction.
   - Upon the confirmation of the transaction, the smart contract function is executed[7] and the blockchain state is updated[8].
6. Termination of the smart contract to stop execution of its functions[9].

Consequently, implementation and testing phase is not trivial and should be taken into account seriously.

To this end, there are various different verification, validation and testing (VV&T) solutions for different purposes proposed in the literature. However, there is no survey covering all the VV&T solutions for smart contracts.

Based on this observation, the objective of this internship is to justify the need to develop an agent-based model to analyse by simulation the *operational safety* from the angle of the availability of the consensus protocol of the Hyperledger blockchain. To this end, various hypotheses will be tested in order to build a solid design for applications that use Hyperledger Fabric. Thus, this study will be divided in three major sections:

- Part I: Identify existing testing/evaluation approaches for smart contracts.
- Part II: Design a smart contract model for Hyperledger Fabric and implement it on a blockchain simulator.
- Paret III: Validate the implemented model through a use case.

[7] In fact, this is the case when the blockchain uses the Order-Execute-Update approach. However, there are also blockchains that use the Execute-Order-Validate-Update approach [3, 38].

[8] The world state can always be regenerated from the blockchain. It works as a cache that only stores the current value of a state.

[9] Here it should be noted that smart contract can not be deleted from the blockchain.

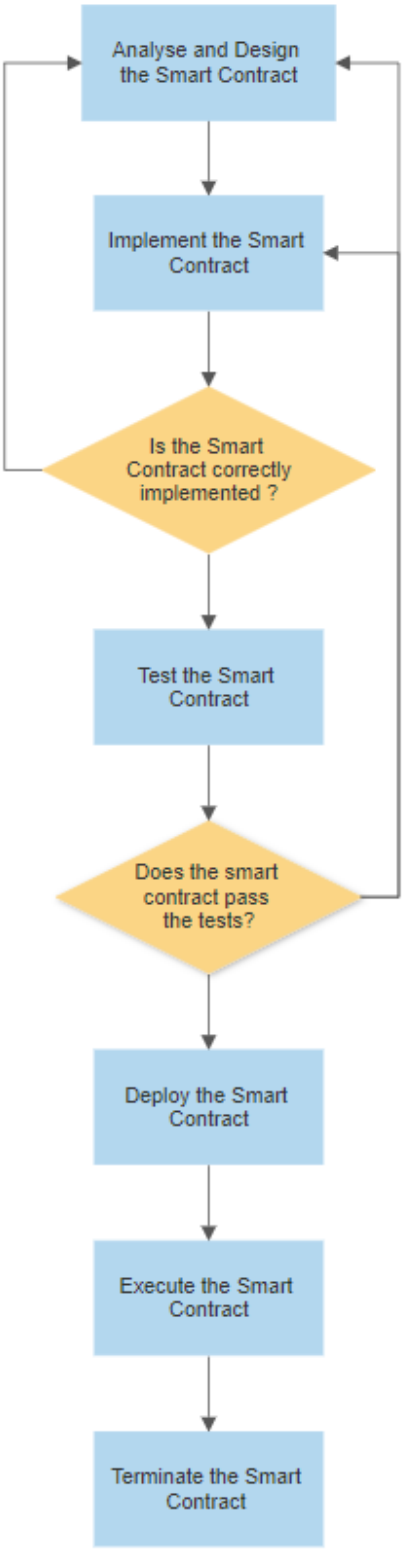


Figure 1: Smart contract lifecycle

2.1 Context

To contextualize, the Dils[10] of List Institute[11] mandates to research and develop the key technologies of digital deployment: methods and tools for designing, producing and validating digital systems. It includes the transfer of know-how and support from internal CEA and external users to the new methods and tools developed by the department. It is composed of five different laboratories as shown in Figure 2:

Figure 2: DILS composition

The internship was carried out at LICIA CEA, a laboratory that develops methods, algorithms and tools for formal analysis and engineering based on agents for: modeling, formalization and implementation of collaborative systems service contracts.

Particular interest is given to the integration of ”smart contracts” for the realization of trust systems; the elaboration and implementation of governance mechanisms through algorithms based on collaborative multi-agent consensus. Particular attention is paid to blockchain mechanisms and incentive mechanisms; the development of analysis techniques based on game theory for assessing the robustness and resilience of large cooperative systems.

Actually, the world of blockchains is in turmoil and many technologies are emerging on a regular basis. The different consensus protocols (proof-of-work, proof-of-stake, PBFT, etc.) are all alternatives and combinations that must be understood, sorted and mastered in order to direct decision-making and ensure the adequacy between the need and the available technology.

In addition, given the lack of feedback that we have on blockchain technology (due to the youth of the technology) the dependability of blockchain applications has yet to be justified, which is a research subject in itself.

In this internship subject we focus on the Hyperledger Fabric blockchain, a permissionned blockchain infrastructure, providing a modular architecture with role-delineation between infrastructure nodes, execution of smart contracts (called chaincode in Fabric), configurable consensus and membership services.

2.2 Contributions

Blockchain and smart contract based solutions can facilitate secure exchanges within a critical application. To do this, it is proposed to provide a simulation model to assist in the analysis, development and implementation of safe protocols and smart contracts.

[10] Département ingénierie logiciels et systèmes
[11] Systems and Technology Integration Laboratory

The objective of this internship is to develop an agent-based model to analyse by simulation the operational safety from the angle of the availability of the consensus protocol of the Hyperledger blockchain. To this end, various hypotheses will be tested in order to build a solid design for applications that use Hyperledger Fabric.

The approach is to understand the Hyperledger Fabric's architecture and make the following contributions:

- Analyze existing approaches and solutions for verifying, validating and testing smart contracts into distinctive groups w.r.t common characteristics.
- Detect limits of available solutions and justify our approach.
- Write and submit a white paper: "A Survey of Verification, Validation and Testing Solutions for Smart Contracts" to BCCA2021 (The Third International Conference on Blockchain Computing and Applications)
- Propose an architecture to simulate smart contracts.
- Implement the proposed architecture for blockchain agnostics first then for the Fabric's one.
- Validate the proposed solution via a representative use case.

3 State-of-the-art: VV&T Solutions

We provide a state-of-the-art of the available solutions that supports smart contracts. This section is organized as follows: First, we divide the VV&T solutions into four distinctive categories and explain them (Section 3.1). Then we analyze each category and compare them with respect to their proprieties (Section 3.2). After, we provide a discussion about how these solutions can be used together (Section 3.3). Finally, we identify open challenges and conclude the first section (Section 3.4).

3.1 Testing

From a software engineering point of view, to ensure the correctness of smart contracts, we use a technique conducted to perform verification, validation and testing (VV&T). Verification is the process to ensure that "we are developing the right product" (*i.e.* it meets the *specifications)*, validation is to ensure that "we have developed the product right" (*i.e.* it fulfills its intended *purpose*) and testing is to reveal "the existence of errors in the product", as stated in [7]. This involves verifying whether the detailed functional VV&T of business logic and process are operating as expected.

Since they are immutable code, once deployed, they are final and cannot be updated. The only way to fix a bug on a deployed smart contract is to deploy a new version of it; the old version remain there forever. So perfoming VV&T before deployment make sure the smart contract has the expected behaviors.

To tackle these problems, many VV&T solutions have already been proposed and are still being proposed in the literature. To better analyze these solutions and compare them, we grouped them into four main categories: public test networks, security analysis tools, blockchain emulators with smart contracts support and blockchain simulators with smart contracts support.

3.1.1 Public Test Networks

Blockchain systems are deployed onto public networks, called main networks. It is the end product available for the public to use. For VV&T purposes, there are also public available networks, called (public) test networks, in which anyone can access via their clients (*i.e.* popular wallet interfaces).

Public test networks enable to compile and deploy smart contracts and perform frequent tasks, such as running tests, automatically checking code for compilation errors or interacting with smart contracts in a public environment.

When using a public test network, the environment conditions are the same as the main network. The main difference is that, while in the main network crypto-currencies are traded for goods and services, in test networks they are given away for free without the need for mining. Hence, users do not have sufficient control to configure the network conditions. They are created and only configured by people from the community.

The main steps to test smart contracts using public test networks are:

- Choose a public test network depending on the desired target blockchain.
- Create a wallet to store the crypto-currencies using a blockchain client.
- Claim a desired amount of coins for free from a *Faucet Website*
- Connect the blockchain client to the test network.
- Write a smart contract using a target language and deploy it onto the test network.
- For each smart contract function, write a test using a language that can interact with the blockchain client and execute it through the blockchain client.

There are several public test networks that exist in the literature. The Ethereum community proposed *Ropsten*[12], *Rinkby*[13], *Kovan*[14] and *Görli*[15] test networks. While the Tezos community introduced Teztnets[16], a platform to help in the deployment of Tezos test networks, namely *Florencenet* and *Granadanet*. Testnet3[17] is another test network proposed by the Bitcoin community, nearly identical in characteristics to Bitcoin blockchain, that overcomes some difficulties related to transaction delay of the previous Testnet2. The Hyperledger Fabric community has also released two versions of *Fabric test network*[18] (v1.4 and v2.1) that enable developers to test their smart contracts and applications. Tendermint community provides gaia and basecoind tesnets for Tendermint/Cosmos-like blockchains.

3.1.2 Security Analysis Tools

Security analysis tools allow automatically analysing smart contracts and detecting some of their *predefined* common vulnerabilities. without *necessarily* deploying the smart contract to a blockchain test network. To validate, verify and test smart contracts using such tools, it is necessary to write the smart contract and either invoke the dedicated tool command or add the tool plugin, if possible, to a blockchain emulator that supports it. The security analysis tools then analyze either the source code of the smart contract or its compiled blockchain virtual machine bytecode. Most of the security analysis tools in the literature are developed for the Ethereum blockchain [2, 4, 19]. The most well-known ones are *Why3* [36], *Oyente* [26], *Mythx*[19], *Mythril*[20], *SmartCheck*[21], *Securify* [42], Osiris [12], Sereum [33], *Manticore*[22], *MAIAN*[23], *Solgraph*[24], *Reguard* [25], Slither [15], *Fether* [44], *FSolidM* [30] , *VeriSolid* [31], *ZoKrates* [14] and *Vandal* [8].

Concerning other blockchain systems, to the best of our knowledge, there are only a few solutions, namely *Chaincode analyzer*[25] [23] and *Revive^^c*[26] for Hyperledger Blockchain, *Zeus* for Ethereum and Hyperledger [22], and *SODA* [11] for any blockchain that adopts EVM as its smart contract runtime.

3.1.3 Blockchain Emulators with smart contract support

Blockchain emulators are software programs that imitate the features of a blockchain. They enable reproducing blockchain networks locally to mimic the outer behavior of the main network features. In other words, they duplicate completely the main network in a virtual environment to emulate VV&T scenarios. Thus, it allows the developer to debug and test smart contracts.

Generally, the emulators come with a default network configuration file but it can be expanded to change the environment options (include more networks, choose the mining mode, adapt the block

[12]Ropsten Test Network, `https://ropsten.etherscan.io/`, last accessed on 26/05/2021.

[13]Rinkby Test Network, `https://rinkeby.etherscan.io/`, last accessed on 26/05/2021.

[14]Kovan Test Network, `https://kovan.etherscan.io/`, last accessed on 26/05/2021.

[15]Görli test network, `https://goerli.net/`, last access on 26/05/2021.

[16]Teztnets,`https://teztnets.xyz/`, last access on 04/06/2021.

[17]Bitcoin actual test network, `https://en.bitcoinwiki.org/wiki/Testnet#Testnet_vs_Testnet2`, last access on 14/06/2021.

[18]Hyperledger Fabric test netxork , `https://hyperledger-fabric.readthedocs.io/en/latest/test_network.html`, last access on 26/05/2021.

[19]Mythx, `https://mythx.io/about/, lastaccess26/05/2021`, last access on 25/05/2021.

[20]Mythril, `https://github.com/ConsenSys/mythril`, last access on 25/05/2021.

[21]SmartCheck, `https://smartcontracts.smartdec.net/`, last access on 25/05/2021.

[22]Manticore, `https://github.com/trailofbits/manticore`, last access on 27/05/2021.

[23]MAIAN, `https://github.com/ivicanikolicsg/MAIAN`, last access on 27/05/2021.

[24]Solgraph, `https://github.com/raineorshine/solgraph`, last access on 27/05/2021.

[25]Chaincode analyzer, `https://github.com/FujitsuLaboratories/ChaincodeAnalyzer`, last access on 10/06/2021.

[26]Revive^^c tool, `https://github.com/sivachokkapu/revive-cc`, last access on 01/06/2021.

time, etc) or smart contracts options.

To test smart contracts on emulators, a user needs to:

- Configure the blockchain if wanted (most of emulators come with a default configurations).
- Write and compile a smart contract.
- Migrate (or deploy) the smart contract locally on to the emulated blockchain.
- Run automated tests or write quick and effective tests.

In the literature, several smart contract emulators exist: *Hyperledger Umbra*[27], *Hyperledger Caliper*[28], *Hardhat*[29], *Truffle*[30], *Brownie*[31], *Takamaka* [41], *Ganache*[32], *Blockbench* [13], *Hawk* [24], *Remix*[33], *Tenderly*[34], and *Embark*[35].

3.1.4 Blockchain Simulators with smart contract support

A blockchain simulator can mimic the internal behaviours of blockchains. They are used when there is the need for the blockchain to perform in an expected way. It helps developers create a copy of an existing blockchain into a virtual environment to get an idea about how, in our case, smart contracts work. It does not follow all the rules of a main network but simulate the relevant behaviors. It enables users design, implement and evaluate a blockchain with the desired configuration of the network's initial conditions using different settings and parameters.

To test smart contracts on simulators, a user should:

- Set up the blockchain configuration
- Write a smart contract
- Deploy the smart contract on the simulated blockchain
- Write tests for deployed smart contracts' functions

There are several blockchain simulators in the literature, yet, to the best of our knowledge, there is only one blockchain simulator supporting smart contracts: *Guantlet*[36] [5].

3.2 Analysis of Solutions

VV&T solutions are used to build smart contracts more efficiently and aim at improving their security and correctness. In this section, we examine and compare the aforementioned solutions.

Table 1 summarizes the most known smart contracts vulnerabilities w.r.t three scopes: (1) smart contract scope that occurs due to the source code (such as *integer overflow/underflow* when performing operations with value limitations), (2) application scope that represents interactions between smart contracts or smart contracts and other entities (such as *Front Running*) and (3) blockchain system

[27] Hyperledger Umbra, `https://github.com/hyperledger-labs/umbra`, last access on 08/06/2021
[28] Hyperledger Caliper, `https://hyperledger.github.io/caliper/`, last access on 27/05/2021
[29] Hardhat test network, `https://hardhat.org/`
[30] Truffle Suite,`https://www.trufflesuite.com/`, last access on 26/05/2021
[31] Brownie framework, `https://eth-brownie.readthedocs.io/en/stable/`, last access on 01/06/2021
[32] Ganache, https://www.trufflesuite.com/docs/ganache/overview, last access on 26/05/2021
[33] Remix: `https://remix.ethereum.org/`, last access on 25/05/2021.
[34] Tenderly: `https://tenderly.co/transaction-simulator/`
[35] Embark tutorial, `https://framework.embarklabs.io/docs/contracts_configuration.html` last access on 09/06/2021.
[36] Guantlelt, `https://gauntlet.network/`, last access on 16/06/2021.

scope that exercises the full blockchain entities togheter (such as *Transaction Order Dependence* when the order of transactions can be easily manipulated) .

Table 3 summarizes the available VV&T solutions for smart contracts w.r.t target blockchains[37], smart contract languages, testing languages, vulnerabilities and Table 2 presents the level of control a developer have while using these different categories. We can easily see that the security analysis tools available are numerous compared to other categories. More precisely, we have 10 public test networks, 20 security analysis tools, 10 blockchain emulators with smart contract support and only 1 blockchain simulator with smart contract support.

Besides, it is also easy to see the vast majority of solutions support Ethereum while support for other blockchains is more scarse. Bitcoin, despite being the most known blockchain, has only one VV&T solution, a public test network that supports smart contracts (i.e. scripts), but there is no tool or emulator available for it.

3.2.1 Target Blockchain

As shown in Table 3, most of the VV&T solutions are dedicated to a very limited number of target blockchains. It is natural that, all public test networks are dedicated to specific target blockchains. Each target blockchain has at least one dedicated public test network. Most security analysis tools, however, including Ropsten, Rinkby, Oyente, Mythril and Smartcheck support only the Ethereum blockchain. There are only two tools supporting Hyperledger. Concerning Tendermint/Cosmos-like blockchains, most security analysis tools are supported, while no tools are available for the other target blockchains.

Blockchain emulators are similar to public test networks and thus they are usually dedicated to a single target blockchain or its derivations. Hardhat, Truffle and Brownie, for instance, support Ethereum and its variations. But, blockchain emulators also sometimes support other blockchains. Hyperledger Caliper and Blockbench, for example, support Ethereum and derivations of Hyperledger.

There is only one simulator solution but it is better at supporting various target blockchains if they are well designed. The simulator identified, Gauntlet, supports several types of target blockchains, namely Ethereum and Tendermint/Cosmos.

3.2.2 Smart Contract and Test Languages

Concerning programming languages, most VV&T solutions support different ones for writing smart contracts and writing tests (Table 3). The test networks for Ethereum support Solidity for smart contracts while tests are written in Javascript. Hyperledger Fabric and Tendermint/Cosmos test networks provide a larger range of languages, such as Solidity, Go, Node.js, Java, Javascript and Python for both smart contracts and tests.

When we look at security analysis tools, since most of them are dedicated to Ethereum, they are mostly supporting Solidity as a smart contract language. Most of these tools do not permit testers to implement their tests and provide predefined commands. Concerning emulators, they are similar to test networks and they support dedicated languages of the target blockchains. Concerning simulators, Gauntlet supports python as programming language.

3.2.3 Vulnerabilities and Attacks

Among all VV&T solutions, only security analysis tools come with pre-defined vulnerability tests that can be directly used (Table 3).

More in detail, all security analysis tools have pre-defined vulnerabilities and it is also possible to improve the supported vulnerabilities using plug-ins (e.g., FSolidM and VeriSolid). Public test

[37]"Target blockchain" covers all the blockchains based on the principle one (*i.e* Ethereum covers Parity, Quorum, etc.)

networks and simulators, in contrast, provide manual VV&T by user-defined scenarios. Emulators provide manual and automated VV&T since they can add security analysis tools as plug-ins. For example, Remix and Truffle can add Mythx as a plug-ins and make use of its pre-defined vulnerabilities.

Regarding the vulnerabilities, the ones that are addressed the most by security analysis tools are Reentrancy, Timestamp dependence, Transaction Ordering Dependence and Integer overflow/underflow as shown in Table 3.

Considering Table 1, other vulnerabilities[38] exist w.r.t different scopes. Security analysis tools mostly focus on the smart contract code and can not perform VV&T from the blockchain system scope as they do not have a global view of it. Considering other approaches, they enable us checking for different scopes of vulnerabilities.

A series of challenges have emerged related to vulnerabilities in the smart contract programming which lead to various attacks [34].

Here is a brief description of some common smart contracts vulnerabilities and attacks that the testing solutions analyze (for more details, refer to Figures 5 and 6 in the Appendix 8) .

Scope / *Vulnerab. & Attacks*	Smart Contract	Application	Blockchain System
Integer Overflow/Underflow	✓	x	x
Integer Bugs	✓	x	x
Unchecked Send Bug	✓	x	x
Mishandled Exceptions	✓	x	x
Gas costly patterns	✓	x	x
tx.origin usage	x	✓	x
Callstack depth exception	x	✓	x
Reentrency	x	✓	x
Front Running	x	✓	x
Timestamp Dependence	x	✓	x
DoS Attack	x	✓	x
DAO attack	x	✓	x
Black listed imports	x	x	✓
Transaction Order Depend.	x	x	✓
Global State Variables	x	x	✓
Call to the unknown	x	x	✓
Read after write	x	x	✓

Table 1: Scopes of vulnerabilities and attacks [23, 32]

3.2.4 Parameter Control

In Table 2, a comparison is made between the four categories considering the configuration parameters in control. For that, we fix a set of network, blockchain and smart contract parameters:

- Network-level paramaters:
  - Size : number of nodes of the network
  - Message transmission delay : communication delay between nodes
  - Message transmission reliability : the ability of a message to be successfully transmitted within its deadline.
- Blockchain-level parameters:
  - Tx Fees : transaction fees
  - BlockSize : the maximum size of blocks

[38] Smart Contract Weakness Classification Registry, `https://swcregistry.io/`, last access on 01/07/2021

  - BatchTimeOut : Block creation frequency.
  - BlockCreationMode : define the mining mode (mine when needed, always mine, turn off the miner, etc.), the consensus to consider, its difficulty,

- Smart contract level parameters:
  - size : the maximum size of a smart contract
  - dependencies : specifies the smart contracts it depends on.

In public test networks and blockchain emulators, blockchain-level parameters (e.g., chain configuration, level of difficulty to mine blocks) are defined in the genesis block. In blockchain simulators, on the other hand, the parameter are usually configured in a configuration file which is used by the simulator to initialize the simulation.

VV&T Solutions			Test Net.	S.A. Tools	Bc. Emu	Bc. Simu
Parameters	*Network*	Size	x	x	Partial	✓
		Msg trans. delay	x	x	Partial	✓
		Msg trans. reliability	x	x	Partial	✓
	Blockchain	Genesis Block	✓	Partial	✓	✓
		Tx fees	x	x	Partial	✓
		Block size	x	x	Partial	✓
		Batch time out	x	x	Partial	✓
		BlockCreation Mode	x	x	Partial	✓
	Smart Contract	Size	x	x	Partial	✓
		Dependencies	✓	✓	✓	✓

Table 2: Control of parameters

3.3 Discussion

Despite the advances in smart contracts VV&T solutions during the last couple of years, our study highlights several open challenges to be tackled by future work. Not surprisingly, the vast majority of solutions support the Solidity language since Ethereum is the mostly supported blockchain. However, tests are usually conducted with general purpose languages like JavaScript, Java or Go.

We identify the following challenges:

3.3.1 Pre-defined vs User-defined vulnerabilities

Most security analysis tools have a useful set of pre-defined vulnerabilities to analyze smart contracts and thus save time of developing user-defined scenarios. However, these tools consider generic properties and do not capture specific vulnerabilities such as the *Front Running* vulnerability which is one of the major security issues. Thus, these tools are effective in detecting typical errors but ineffective in detecting atypical vulnerabilities.

Therefore, manual generation of meaningful scenarios becomes a necessity. It enables us inspect deep corner cases that users are already aware of, given the wide range of vulnerabilities that can occur in smart contracts. Still, developing them could be very costly in time.

The appropriate way to proceed is a mix between pre-defined and user-defined solutions, which will speed up the VV&T process. In that way, we do not waste time in writing scenarios that security analysis tools can already detect and only focus on detecting the non-defined ones.

	Target Blockchain					SC Language						Test Language						Vulnerabilities					
																		Pre-defined					
	Bitcoin	*Ethereum*	*Hyperledger Fabric*	*Tendermint/Cosmos*	*Others*	*(Bitcoin) Script*	*Solidity*	*Go*	*Java*	*JavaScript*	*Others*	*Solidity*	*Go*	*Java*	*JavaScript*	*Python*	*Others*	*Reentrency*	*Trans. order depen.*	*Timestamp depen.*	*Int. over-/under-flow*	*Others*	*User-defined*
Public Test Networks																							
Ropsten	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	x	✓
Rinkby	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	x	✓
Görli	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	x	✓
Testnet3	✓	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	x	x	x	x	x	✓
Florencenet	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	✓
Granadanet	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	✓
Cardano Testnet	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	✓
Hyperledger Testnet	x	x	✓	x	x	x	x	✓	✓	✓	✓	x	✓	✓	✓	x	✓	x	x	x	x	x	✓
Gaia	x	x	x	✓	x	x	✓	✓	✓	✓	✓	x	✓	✓	✓	x	✓	x	x	x	x	x	✓
Basecoind	x	x	x	✓	x	x	✓	✓	✓	✓	✓	x	✓	✓	✓	x	✓	x	x	x	x	x	✓
Total	1	3	1	2	3	1	5	3	3	3	6	0	3	3	6	0	7	0	0	0	0	0	10
Security Analysis Tools																							
Oyente	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	✓	✓	x	✓	x
Mythx	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	x	x	✓	✓	x
Mythril	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	✓	✓	✓	✓	x
SmartCheck	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	✓	✓	✓	✓	x
Securify	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	✓	x	x	✓	x
Osiris	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x
Sereum	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	x	x	x	✓	x
Manticore	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	x	✓	✓	✓	x
MAIAN	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x
Solgraph	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x
Reguard	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	x	✓	x	✓	x
Slither	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓	x	✓	x	x	x
Fether	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x
FSolidM	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	✓
VeriSolid	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	✓
Zokrates	x	✓	x	✓	x	x	✓	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓	x
Vandal	x	✓	x	✓	x	x	✓	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	✓	x
Chaincode Analyzer	x	x	✓	x	x	x	x	✓	x	x	x	x	✓	x	x	x	x	x	x	✓	x	✓	x
Zeus	x	✓	✓	✓	x	x	✓	✓	✓	✓	x	x	x	x	x	x	✓	✓	✓	x	✓	x	x
SODA	x	✓	x	✓	x	x	✓	x	x	x	x	x	✓	x	x	x	x	✓	x	✓	x	✓	x
Why3	x	✓	x	x	x	x	x	x	x	x	✓	x	x	x	x	x	✓	x	x	x	x	x	✓
Total	0	19	2	19	0	0	19	2	1	1	0	1	2	0	0	0	1	10	5	8	5	18	0
Blockchain Emulators with smart contract support																							
Hyperledger Umbra	x	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	✓
Hyperldeger Caliper	x	✓	✓	x	✓	x	✓	x	✓	✓	x	x	✓	✓	✓	x	x	x	x	x	x	x	✓
Hardhat	x	✓	x	x	x	x	✓	x	x	x	x	✓	x	x	✓	x	x	x	x	x	x	x	✓
Truffle	x	✓	x	x	x	x	✓	x	x	x	x	✓	x	x	✓	x	✓	x	x	x	x	x	✓
Brownie	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	✓	x	x	x	x	x	✓
Ganache	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	✓	x	x	x	x	x	x	x	✓
Blockbench	x	✓	✓	x	x	x	x	✓	✓	✓	✓	x	✓	✓	✓	x	x	x	x	x	x	x	✓
Remix	x	✓	x	x	x	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	x	x	x	✓
Tenderly	x	✓	x	x	x	x	x	x	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	✓
Embark	x	✓	x	x	x	x	x	x	x	✓	x	x	x	x	✓	x	x	x	x	x	x	x	✓
Total	0	9	3	0	1	0	6	2	2	4	1	3	2	2	7	1	2	0	0	0	0	0	10
Blockchain Simulators with smart contract support																							
Gauntlet	x	✓	x	✓	✓	x	x	x	x	x	✓	x	x	x	x	✓	x	x	x	x	x	x	✓
Total	0	1	0	1	1	0	0	0	0	0	2	0	0	0	0	1	0	0	0	0	0	0	2
Total	1	32	6	22	5	1	30	7	6	8	9	4	7	5	13	2	10	10	5	5	4	18	22

Table 3: Resume of VV&T solutions wrt the target blockchain, smart contract language programming, test language programming and type of vulnerabilities detected.

3.3.2 Programming Languages

As demonstrated in Table 3, most testing solutions support a domain-specific language (DSL), Solidity, as a smart contract language. Others (most recent ones) use common general-purpose languages such as Java that were not initially conceived for smart contracts and thus make less expressive but easier to developers since they are more familiar with them. Hence, the vulnerabilities and risks associated to a certain language are not the same with others, due to each language features and restrictions. With DSLs, very useful libraries that facilitate the programming of smart contracts are provided. While with general-purpose languages, there are no features or restrictions to write safe smart contracts. For example, using Go language, a non determinism may arise, leading to random number generation or global variables vulnerabilities. This does not mean that with general-purpose languages, more vulnerabilities will occur but using the available libraries in DSLs (*i.e* Solidity provides reusable behaviors and implementations of various standards such as OpenZeppelin, HQ20, DappSys), a consequent number of them can be avoided due to specific restrictions.

3.3.3 Community Participation

A smart contract project can be more successful if a community of volunteers who are globally distributed participate to the VV&T process through the Internet, as also identified by the open source community [39]. This enables the community to make more tests and thus helps in detecting a large range of vulnerabilities. There are two ways to this: (1) the smart contract can be deployed onto a public test network and thus made available to the community, or (2) the smart contract project can be made available to community through a public repository.

3.3.4 Confidentiality

There is a big interest from industries to adapt their existing services or new services as smart contracts. To this end, they are actively doing several proof-of-concept studies. During these studies, the industries usually use confidential information (e.g. sensitive data, confidential algorithms, solutions) that they do not want them to be shared publicly until they finalize their smart contract based solution. In such cases, obviously public test networks are not suitable as a VV&T approach. It is rather better using the other three approaches (i.e. security analysis tools, blockchain emulators or blockchain simulators) without putting their smart contract in a public environment. The disadvantage of this is that they cannot benefit from the community involvement that can speed up vulnerability exploration.

3.3.5 Flexibility of parameters

An important point to mention about these testing solutions is the network size. Actually, testing solutions have generally fewer nodes than a main network. For example, the size of the main Bitcoin blockchain on May 20, 2021 is about 337 Gigabytes[39] whereas the size of the Testnet3 blockchain in 2020 is only about 32 Gigabytes. This makes transactions lighter and faster which simplifies performing attacks as it does not require large capacity. Also, depending on the control parameters, we could perform different tests, whether on vulnerabilities, latency, performance, etc.

3.3.6 Levels of testing

In traditional testing literature, the major levels of testing are unit-, integration- and system-levels, and some type of acceptance-level [10]. Basically, unit testing is used to make sure that the implemented code meets the user specifications and works properly. Integration testing, however, is used to

[39]Size of Bitcoin blockchain on May 20,2021, `https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/`, last access on 14/06/2021

combine different parts and test them jointly (in our case, it could be different smart contracts, or the smart contract and the user interactions). Finally, system testing is used to test all the components to ensure the overall right functioning.

Considering smart contracts, the scopes of potential vulnerabilities are identified in Table 1. Based on this observation, we can say that smart contract, decentralized application and blockchain system level vulnerabilities should be tested using unit, integration and system level testing respectively.

As shown in Table 3, different testing solutions provides mechanisms for testing smart contracts against different types of vulnerabilities and attacks. It is up to the developer/tester to identify the tests to be accomplished and to choose the appropriate testing solutions.

For instance, as shown in Section 3.2, except for simulators, all existing solutions are dedicated to a single blockchain technology and its variations. Consequently, if we want to explore what-if scenarios for different blockchains in order to chose a target platform, obviously simulators are the best option to do so. However, to facilitate such an exploration, the simulator should provide some built-in capabilities that can be applied to various types of blockchain systems. Currently, there is only Gauntlet that can simulate different types of blockchain systems supporting smart contracts.

3.4 Challenges

Smart contracts benefit from a widespread interest because of their huge potential. However, in order to effectively build smart contracts, they need to be tested in an effective and systematic manner. In our state-of-the-art, we analyzed four categories of smart contract testing approaches: using test network, using security analysis tools, using blockchain emulators with smart contracts support and using blockchain simulators with smart contract support.

This shows that several VV&T solutions exist, ensure the correctness and non vulnerable patterns in smart contracts. Nevertheless, they either focus on specific security aspects, a specific blockchain and a programming language or provide limited configurations.

Besides, solutions that limit the implementation language or the environment are less general and require the user to adapt their applications which could be infeasible or very expensive. In addition, the variety of testing solutions available make it confusing and difficult as to where to start working with them.

As a result, we focus on developing a smart contract simulator since it offers more flexibility and control over the system's parameters and enables all the testing levels. And to be more specific, we centralize our project on the Hyperledger Fabric blockchain which offers a new architecture presented in Section 4.1.

This state-of-the-art is submitted to the BCCA 2021 conference (under review).

4 Background

4.1 Blockchain systems

Blockchain is an append-only distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way. Blockchain is an ordered list of blocks - data containers- that are created by nodes participating in the consensus process: they contain recorded data from previous transactions that can not be deleted, and anyone can read it from and verify its correctness. After recording recent transactions, a new block is generated and the blocks will be validated by the miners by analyzing the complete history of the block. If the block is validated, it will be time-stamped and added to the blockchain. Once added, it cannot be modified or deleted. The chain begins with a genesis block at index 0 and each block appended links to its direct predecessor forming the chain.

An appealing application that can be deployed on top of blockchain is smart contracts.

4.1.1 Smart contracts

Smart contracts, proposed in the early 1990s, enable parties to formally specify a cryptographically enforceable agreement, portending Bitcoin's scripting capabilities. A smart contract is a digital contract with an agreement between two people in the form of computer code. They run on the blockchain, so they are stored on a public database and are immutable as the blockchain.

The transactions that happen in a smart contract are processed by the blockchain when the conditions in the agreement are met which means they can be sent automatically without a third party.

More specifically, a smart contract is a deterministic program stored as executable bytecode on the blockchain, which means they inherit certain properties - immutability and global distributability.

They are used for two purposes:

- Hold funds and state, which are stored in the blockchain under the contract's address.
- Run logic/code that performs actions with those funds or updates the contract's state.

To do so, the code itself is replicated across multiple nodes of a blockchain and, therefore, benefits from the security, permanence and immutability that a blockchain offers. That replication also means that as each new block is added to the blockchain, the code is, in effect, executed. If the parties have indicated, by initiating a transaction, that certain parameters have been met, the code will execute the step triggered by those parameters. If no, such transaction has been initiated, the code will not take any further steps.

Before a compiled smart contract actually can be executed on certain blockchains, an additional step is required, namely, the payment of a transaction fee for the contract to be added to the chain and executed upon. In the case of the Ethereum blockchain, smart contracts are executed on the Ethereum Virtual Machine (EVM), and this payment, made through the ether cryptocurrency, is known as "gas."

Smart contracts are first compiled and converted into bytecode before the deployment. Various high-level languages in the community compiles and converts the code to the EVM bytecode. The most popular one is Solidity. This bytecode is then stored on the blockchain and an *address* is assigned to it.

An important point to mention is that smart contracts get executed only due to a transaction call. A contract can call another one but the first contract must be called by a transaction.

To understand more the logic of smart contracts, we give an overview of the execution of a transaction using a smart contract considering Ethereum blockchain and using Remix IDE: A think to mention is, while using Ethereum blockchain, each transaction execution consumes a certain amount of *gas*.

Design

We first design and write the Ethereum smart contract in Solidity as follows 3:

```
pragma solidity >=0.7.0 <0.9.0;

/**
 * @title sampleContract
 * @dev Emit an event that echoes the input message
 */
contract sampleContract {
    event Echo(string message);

    function echo(string calldata message) external {
        emit Echo(message);
    }
}
```

Figure 3: Smart Contract solidity code

Compile

As soon as the contract is compiled (see Figure 4), a file including the contract bytecode (sample-Contract.json required for deployment) and the contract ABI will be generated (see Figure 5).

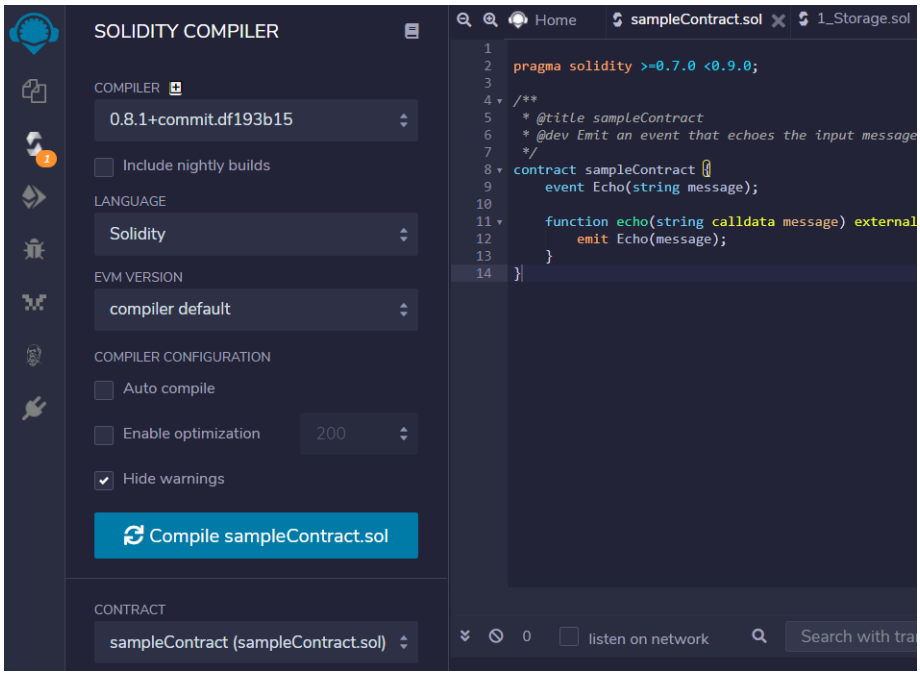


Figure 4: Smart Contract compiled

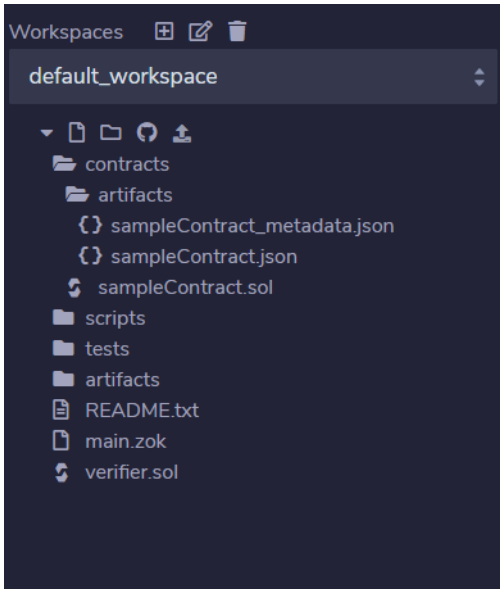


Figure 5: Smart Contract bytecode generated

Deploy

Deploying a smart contract on the Ethereum blockchain is done by the means of the creation of a transaction sent to the special contract creation *address*. Each smart contract is then identified by an *address*, which is derived from the contract creation *transaction* as a function of the originating an account and a nonce. It can be used in a transaction as a sender, a receiver or a callee of the contract's functions.
Once deployed, we can see that the contract has an address in Figure 6. The smart contract is then stored on the blockchain.

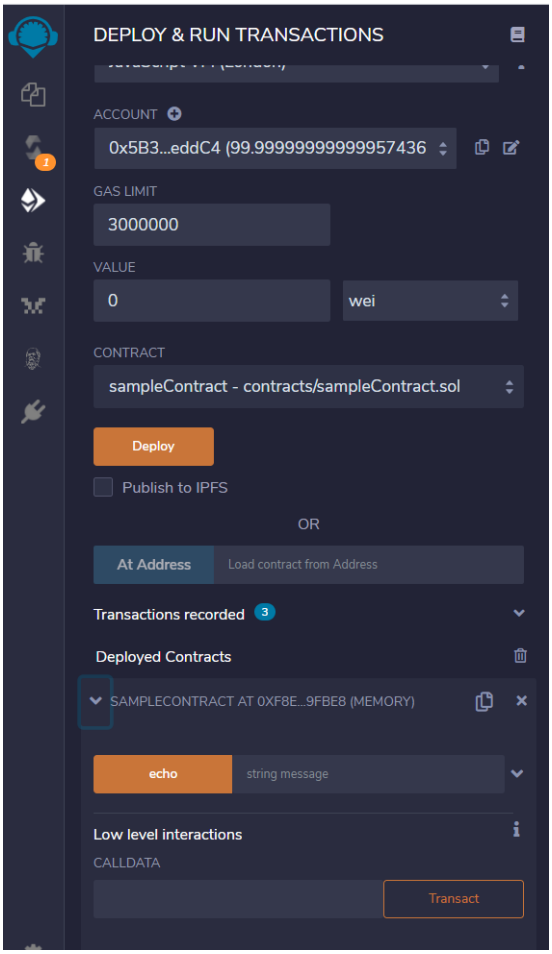


Figure 6: Smart Contract deployed

Smart contract calls

Once the smart contract is deployed, users can call its functions using transaction/message calls. Users or contracts can call other contracts by sending *message* calls that determine a sender, a receiver, a *data payload*, value of Ether sent and the amount of gas left and the type data to return.

Having access to the payload, the smart contract executes the method/function then returns the result which will be stored at the sender's memory.

The payload is composed of:

- *Function Selector*: to identify the function to execute
- *Function Arguments*: to determine the arguments of the function to call

Once the function call is executed, we can observe the balance changes. The list of functions from the smart contract can be seen in 6. One can interact with a deployed smart contract using the function buttons. Once we call a smart contract function, we can see the status, the cost, the address of the transaction and more as in Figure 7.

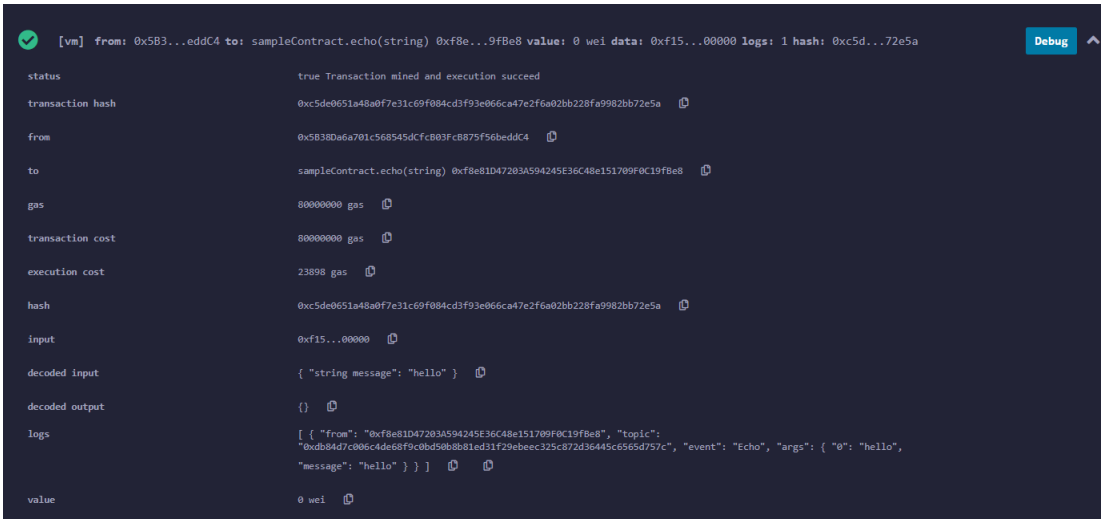


Figure 7: Smart Contract call

4.1.2 Blockchain Architectures

A transaction is an operation that requires changing the status of the blockchain, a message that is sent from one account to another account that includes *payload* (binary data) and an amount of cryptocurrencies.

Each time transactions are created, they follow a certain steps depending on the blockchain architecture. Actually, nowadays, two blockchain architectures exist:

- An *Execute-After/Order-Execute architecture* (see Figure 8) where we first reach a consensus on the order of transactions and then the transactions are executed by all nodes in order. At the same time, each node participating in mining must "execute" each transaction that is about to be wrapped in the block by itself before "ordering", and discard the invalid transaction. So all nodes "re-execute" the transactions in sequence in order to validate the result after receiving the block.

  The local ledger is updated after the re-execution.

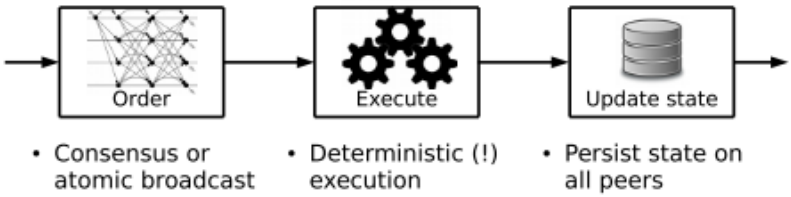


Figure 8: Order-execute architecture [3]

  Bitcoin, Ethereum and Quorum are based on execute-order architecture. In this mode, the node can only update its own state through the repeated execution results of the computation process. It must be repeatedly executed by each node, which makes Nakamoto blockchain inefficient.

  Some of the limitations that this introduces a sequential execution of all transactions which directly affects transaction throughput. It also limits the scalability and endorsement by all peers.

- An *Execute-First/Execute-order-validate* architecture has been proposed in Hyperledger Fabric[3] to support parallel transactions and improve the blockchain's throughput. In this architecture,

transactions are executed before the blockchain reaches consensus on their place in the chain, as illustrated in Figure 9.

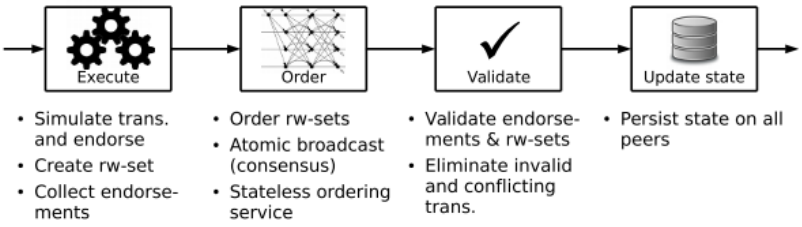


Figure 9: Execute-order-validate architecture [3]

Specifically, the update of the ledger follows three steps, which ensures that all the peers in a blockchain network keep their ledgers consistent with each other:

- **Transaction Execution:** In the first phase, the transaction proposal is sent to each of the required set of peers. Later, the peers independently executes the transaction proposal, simulate it, determine if it is valid and sign it.
- **Transaction Ordering through a consensus protocol :** The second phase is the packaging phase where a specific nodes collaborate to reach consensus, create blocks and order them in a well-defined sequence.
- **Transaction Validation :** In the final phase, the transaction is dispatched to all peers. Once the peers receive the blocks, they ensure that all block transactions have been validated. Then, they commit it to the global state of the blockchain.

Transactions, once executed, the changes in the global state (contracts, accounts, etc.) are recorded only if all execution terminates successfully.

4.2 Agent Based Modeling of Blockchain Systems

Agent-based modeling (ABM) [16] is a powerful simulation modeling technique that excels in its ability to simulate complex systems. Such systems often self-organize themselves and create emergent order. Agent-based models also include models of behaviour and are used to observe the collective effects of agent behaviours and interactions. They have been used to simulate a variety of social phenomena and environments, with the objective of providing decision support for decision makers in various domains.

In ABM, a system is modeled as a collection of autonomous decision-making entities called agents. Each agent individually assesses its situation and makes decisions on the basis of a set of rules. ABM can exhibit complex behavior patterns and provide valuable information about the dynamics of the real-world system that it emulates. It also provides a natural framework for tuning the complexity of the agents: behavior, degree of rationality, ability to learn and evolve and rules of interactions.

The benefits of ABM over other modeling techniques can be captured in the capability of formulating truly flexible actor behavior, capturing emergent phenomena and providing a natural description of a system.

4.2.1 Agent/Environment/Role model

The Agent/Environment/Role (AER) model is based on three first-class abstractions: agent, environment and role. Those abstractions are composable and interact with each other.

To create an agent-based model, the following three elements [21] have to be explicitly dealt with:

- A set of agents, their attributes and behaviours: which are objects representing the active and communicating entities playing roles within environment and play at least one role in an environment.
- Roles, a set of agent relationships and methods of interaction : allowing agents to interact with each other and manipulate passive environment's objects. Depending on the application, agents can communicate either directly by sending a message or through a shared memory or alternatively indirectly as for bio-inspired multi-agent systems by sending signals via the environment. In some applications, agents do not communicate with each other but interact through the use of game theory or learning by observing the behaviour of others.
- The agents' environment : "is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources" [37]. It identifies contexts for patterns of activities (i.e. roles).

To begin the simulation, first, roles of the system to model should be defined. Once done, we define actions/behaviors corresponding to the different roles defined. Then, we define the environment with the allowed roles. Then we define the agent types, their roles and the actions to take during the simulation to interact with one another inside an environment.

4.2.2 Blockchain roles

Considering [1], agents roles in blockchain systems can be resumed into nine generic roles identified in Figure 10.

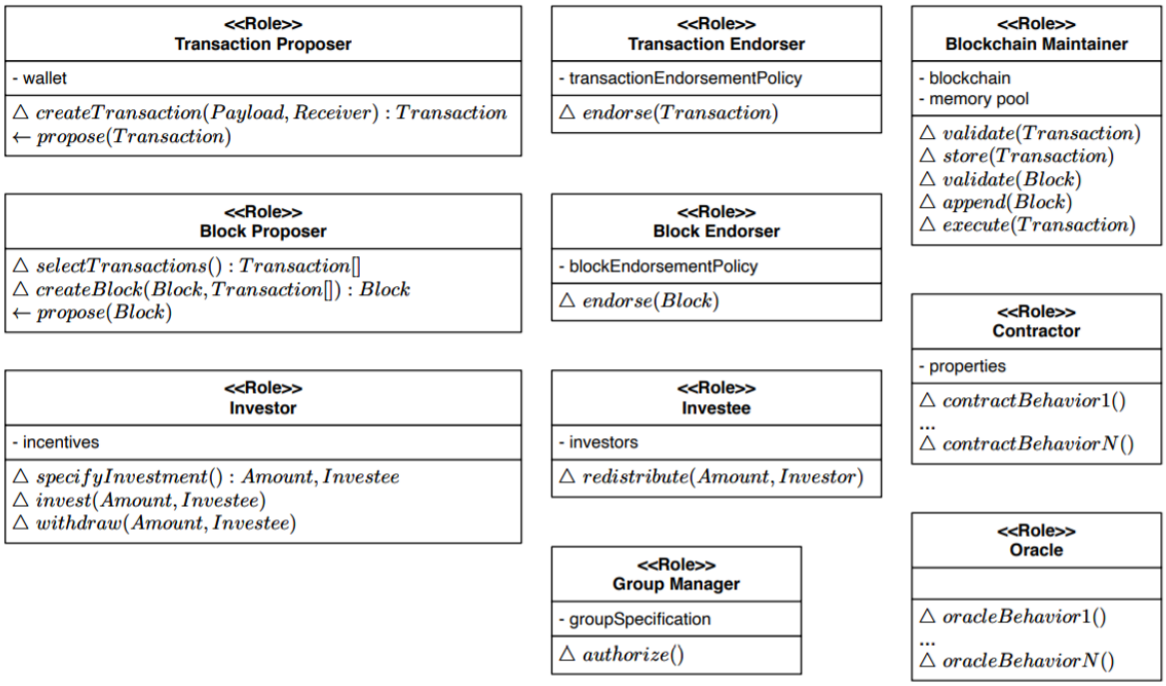


Figure 10: Generic blockchain roles [1]

- Transaction Proposer : in charge of proposing transactions.
- Transaction Endorser : in charge of endorsing the transaction propositions.

- Blockchain Maintainer : in charge of maintaining the data structure.
- Block Proposer : in charge of creating and proposing blocks.
- Block Endorser : in charge of endorsing the proposed blocks.
- Contractor : in charge of fixing the contractual rules between participants.
- Investor : in charge of making investments.
- Investee : in charge of redistributing rewards to the investors.
- Group Manager : in charge of checking the conformity of the structure.
- Oracle : in charge of other services.

For more details, refer to [1].

5 Hyperledger Fabric

Fabric is a framework for developing Blockchain-based solutions for the enterprise, originally contributed by Linux Foundation and IBM, providing a modular architecture with a delineation of roles between the nodes in the infrastructure, execution of chaincode/smart contracts, configurable consensus and membership services.

5.1 Fundamental Elements

The most important components of a Hyperledger Fabric system are (refer to Figure 11):

- *Endorsing/Validating peers*: execute/endorse a given transaction and satisfy the transaction's endorsement policy.
- *Chaincode* : is the *smart contract* that runs on the peers and creates transactions. More broadly, it enables users to formally specify an enforceable agreement on some application logic.
- *Ordering service nodes (OSN)* : provides a shared communication channel to clients and peers, offering a broadcast service for messages containing transactions.
- *Membership service provider (MSP)* : is responsible for managing identities of all participants in the network.
- *Channels* : is defined by a set of members, peers, chaincode and orderer nodes.
- *Applications*: are used by users to interact with a blockchain network by submitting transactions to a ledger or querying ledger content.
- *Organisations*: are entities which have access to channels and can issue identities to participants so that every transaction's source is clear and identifiable.
- *Certification Authority (CA)*: provides a number of certificate services to users of a blockchain.

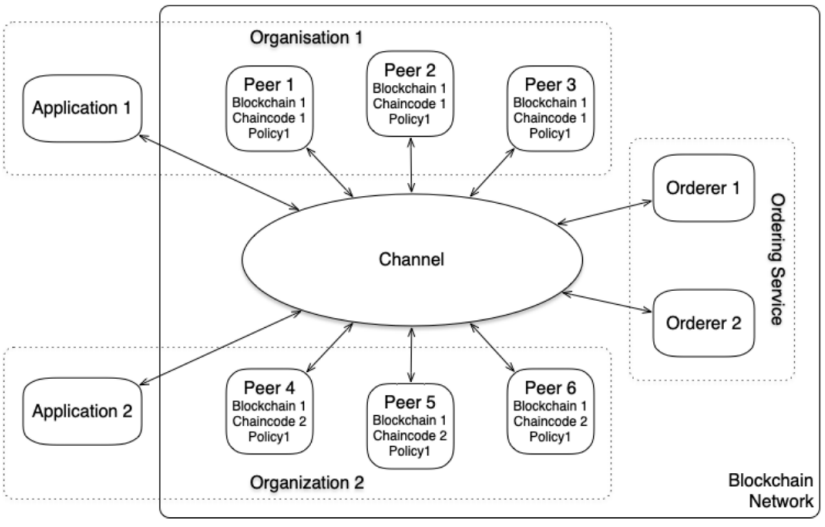


Figure 11: Hyperledger Fabric Elements

5.2 Endorsement Policies

In Hyperledger Fabric, each chaincode/smart contract get validated by the mean of an endorsement policy defined in it and agreed by the channel members. This endorsement policy determines the endorser peers where each one of them must approve or not the result of a transaction execution. To validate a chaincode call, a sufficient number of channel members should approve it. If the chaincode call (transaction call) is validated, it is added to the memory pool, if not, it is not considered.

5.3 Smart Contract Transaction Lifecycle

Hyperledger Fabric follows an Execute-Order-Validate philosophy (refer toSection 4.1.2) while most of the existing blockchains implement an Order-Execute model, like Bitcoin and Ethereum.

For more clarity, an example is provided in Figure 12 that describes the life cycle of a transaction in Hyperledger Fabric blockchain. It is assumed that a channel is configured and working. The chaincode is installed on peers and deployed on the channel. The chaincode contains a logic defining a set of transaction instructions. An example of an approval policy has also been defined for this channel, indicating that Peer P1 and Peer P2 must approve any transaction.

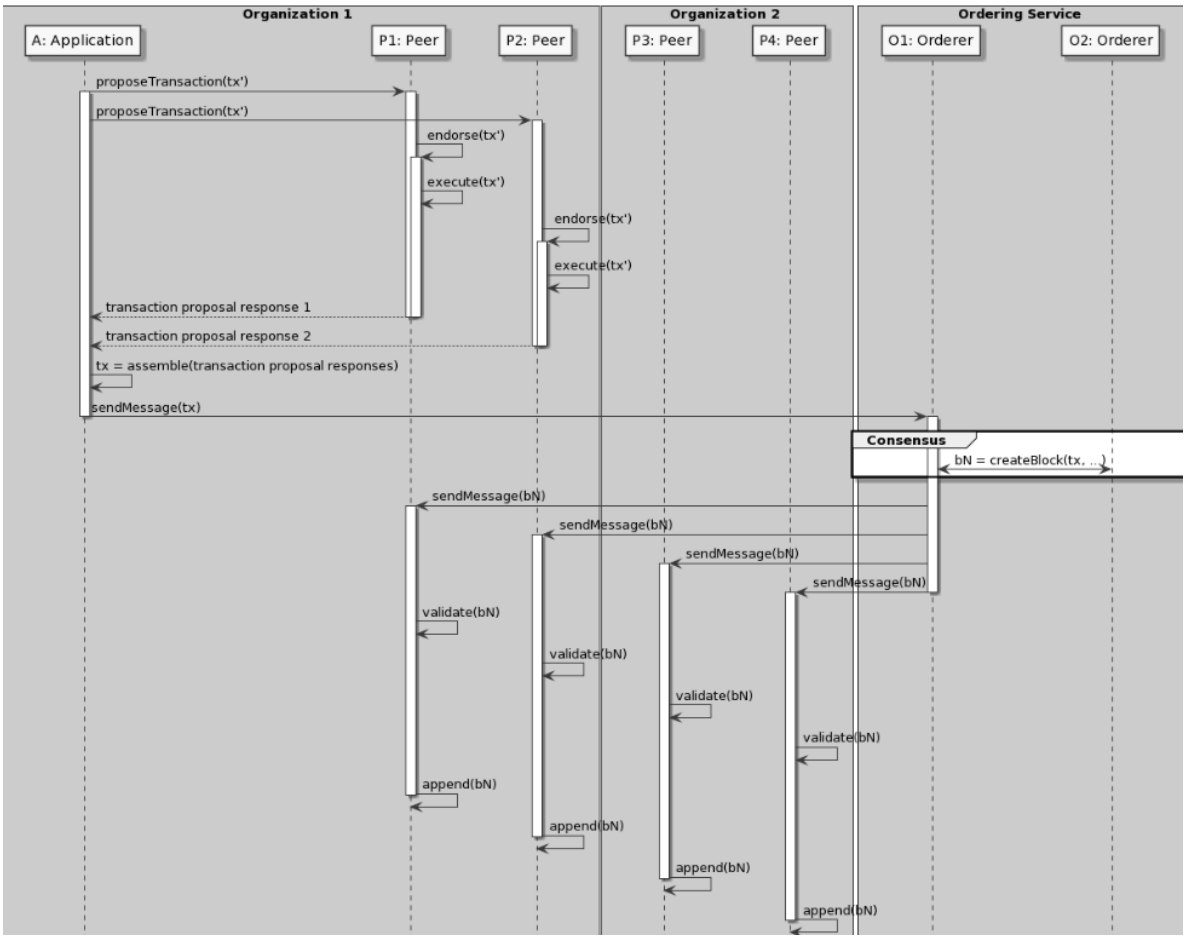


Figure 12: Hyperledger Fabric Transaction Life-cycle

1. Application A is sending a transaction proposal by invoking a chaincode function to the target peers to be endorsed.
2. Approving peers simulates the transactions on its state database, verify signature and then execute the chaincode.

3. Application A verifies peer signatures approving and inspects proposal responses. Then, he broadcasts transaction proposal and response within a “transaction message” to the OSN.
4. The OSN receives transactions from all the network, orders them and creates blocks of transactions by channel. Then the transaction blocks are broadcasted to all peers.
5. Peers validate transactions by ensuring that the approval policy is fulfilled.
6. Finally, each peer appends the block to the channel’s chain.

6 The Proposed Agent-based Simulation Model

The usual stack of a blockchain is divided in three levels as shown in Figure 13:

- Decentralized Application: where data/information is stored into the blockchain and are interpreted directly.
- Blockchain: where new blocks are appended considering the consensus.
- Network: where nodes communicate with each other (topology, messages, ...) is considered.

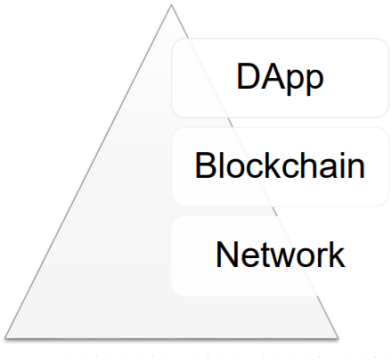


Figure 13: Implementation levels

In our solution, smart contracts are implemented considering the first-level implementation: Decentralized Application considering the Agent/Environment/role model. The environment will manage the nodes and the smart contracts through the use of agents with different roles in order to simulate the blockchain entities, roles and behaviors.

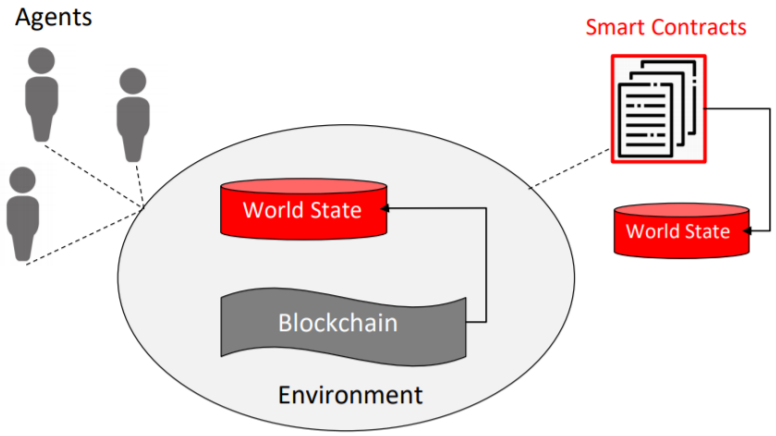


Figure 14: Interactions between Nodes and Smart contracts

The aim is to implement the basic functionalities of smart contracts considering *Nodes*, *SmartContract* agents and *BlockchainEnvironment* (see Figure 14) in light of two architectures: *Execute-After* and *Execute-First*.

In this model, we consider that *Node* agents can only communicate with smart contract agents using *tamper resistant messaging* via the *BlockchainEnvironment*, then the smart contract executes the function called and update the blockchain and the smart contract world state (see Figure 15).

We do not take into account the different steps to execute a transaction (i.e. consensus, endorsement). Instead, we just focus on the update state step.

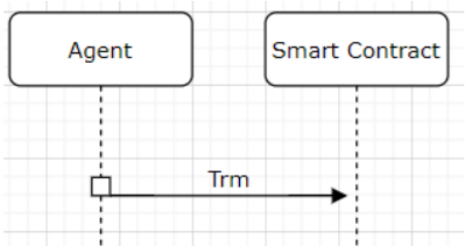


Figure 15: Communication via Tamper Resistant Messages

6.1 Entities, Roles and Behaviors

Following the methodology represented in Section 4.2.1, we define roles, actions, environments and agents successively.

Relying on Section 4.2.2:

6.1.1 Roles

- RNode: is an actor role to represent the transactionProposer role
- RSmartContract: is an actor role to represent the contractor role.

6.1.2 Actions

- ACCreateSmartContract is an action that enables the environement to create blocks each 10 ticks (1 tick equal 1 min).
- ACDeploySmartContract is an action that enables deploying an instance of a smart contract agent
- ACCallSmartContract is an action that enables calling smart contracts methods/functions.
- ACExecuteTxAfter is an action that enables executing transactions depending on the execute-after architecture. In this case, the transactions are executed after creating a block containing them.
- ACExecuteTxFirst is an action that enables executing transactions depending on the execute-first architecture. In this case, the transactions are executed instantly and are added to the memory pool only if they are validated. In our simulation, we consider that a transaction is valid if it does not throw an exception. In the other case, we do not take it into account.

6.1.3 Environment

- BlockchainEnvironment is an simulated environment that plays the BlockchainProposer, Group-Manager and BlockchainMAintainer roles.

6.1.4 Agents

- Node is a simulated agent that plays the RNode role.
- SmartContract is a simulated agent that plays the RSmartContract role.

A description of the project implementation is detailed in the class diagram in Figure 16.

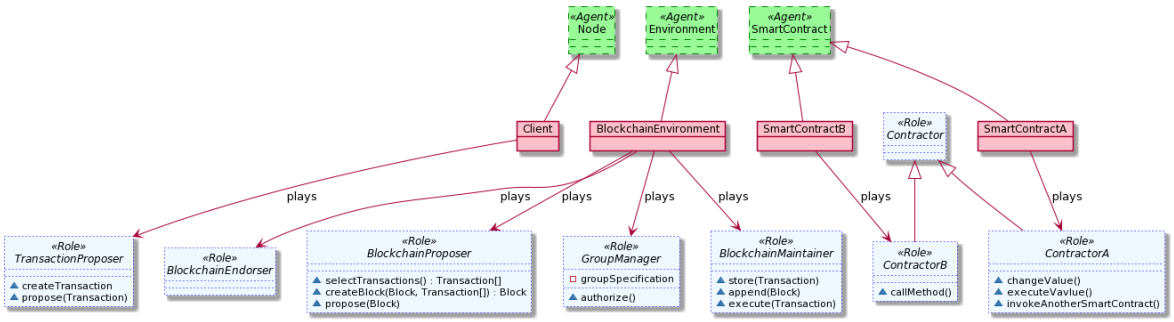


Figure 16: Smart Contract Class Diagram

6.2 Execute-After architecture

In this architecture, the *BlockchainEnvironment* creates blocks each 10 ticks and then execute the transactions in the block.

Reflection API

To invoke the *SmartContract* methods, we use *Reflection API*, a java class that enables examining and modifying the structure and behavior of a class.

The methods used for our project are in Table 4.

Method	Description
public Object newInstance() throws InstantiationException, IllegalAccessException	creates new instance.
public Method getDeclaredMethod(String name,Class[] parameterTypes) throws NoSuchMethodException, SecurityException	returns the method class instance.
public Mehod Invoke(Object, Object[]) thraws TargetException, ArgumentException, TargetInvocationException, MethodAccessException	calls the method represented by the current instance, according to the parameters specified.

Table 4: Reflection API methods

Deployment workflow

Figure 17 demonstrates the lifecycle of deploying the smartContractA:

- The *Client* node:
  - Constructs the payload:
    scCall = createSmartContractCall(SmartContractA.class, ”deploy”, null)
  - Creates the transaction :
    tx = new Transaction(node, smartContractA, scCall)

  - Creates the message proposal:
    proposeMsg = createProposeessage(node,smartContractA, tx)
  - Sends the proposeMsg to the *Blockchain Environment*

- The *Blockchain Environment*:
  - Stores the proposeMsg payload in the *memoryPool*
  - Each 10 ticks :
    * Creates a new block by selecting the transactions from the *memoryPool*, creating a new block and adding it to the blockchain.
    * After the creation of the block, for each transaction in the block:
      · Recover the *scCallRequest*,the scClass, scMethodToInvoke, the nodeAddress
      · If the scMethodToInvoke = ”deploy”, *SmartContractA* is deployed and its address is returned to the *client* node.

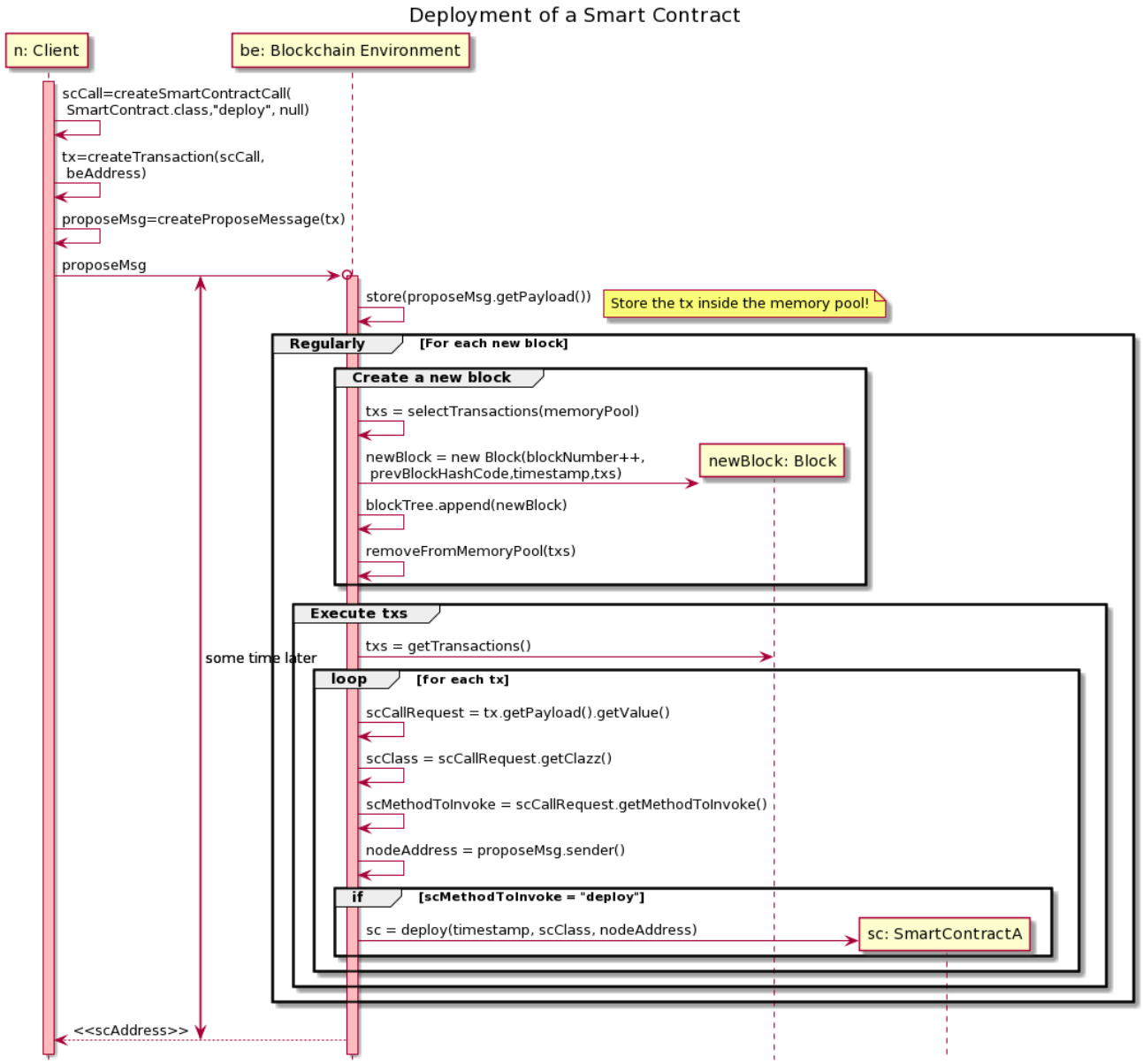


Figure 17: Smart Contract deployment workflow : UML Diagram

Call methods workflow

The UML diagram in Figure 18 demonstrates the lifecycle of invoking a smart contract methods.

- The *Client* node:
  - Contructs the call: scCall = createSmartContractCall(SmartContractA.class, ”deploy”, null)
  - Creates the transaction : tx = createTransaction(scCall, beAddress)
  - Creates the message proposal: proposeMsg = createProposeessage(tx)
  - send the proposeMsg to the *Blockchain Environment*
- The *Blockchain* Environment:
  - Stores the proposeMsg payload in the *memoryPool*
  - Each 10 ticks :
    - Creates a new block by selecting the transactions from the *memoryPool* and adding it to the blockchain.
    - After the creation of the block, for each transaction in the block:
      - Recover the *scCallRequest*, scClass, scMethodToInvoke, scMethodArgs and the receiver.
      - If the scMethodToInvoke != ”deploy”, we invoke the smartContract method.
    - The *SmartContractA* executes the method called.

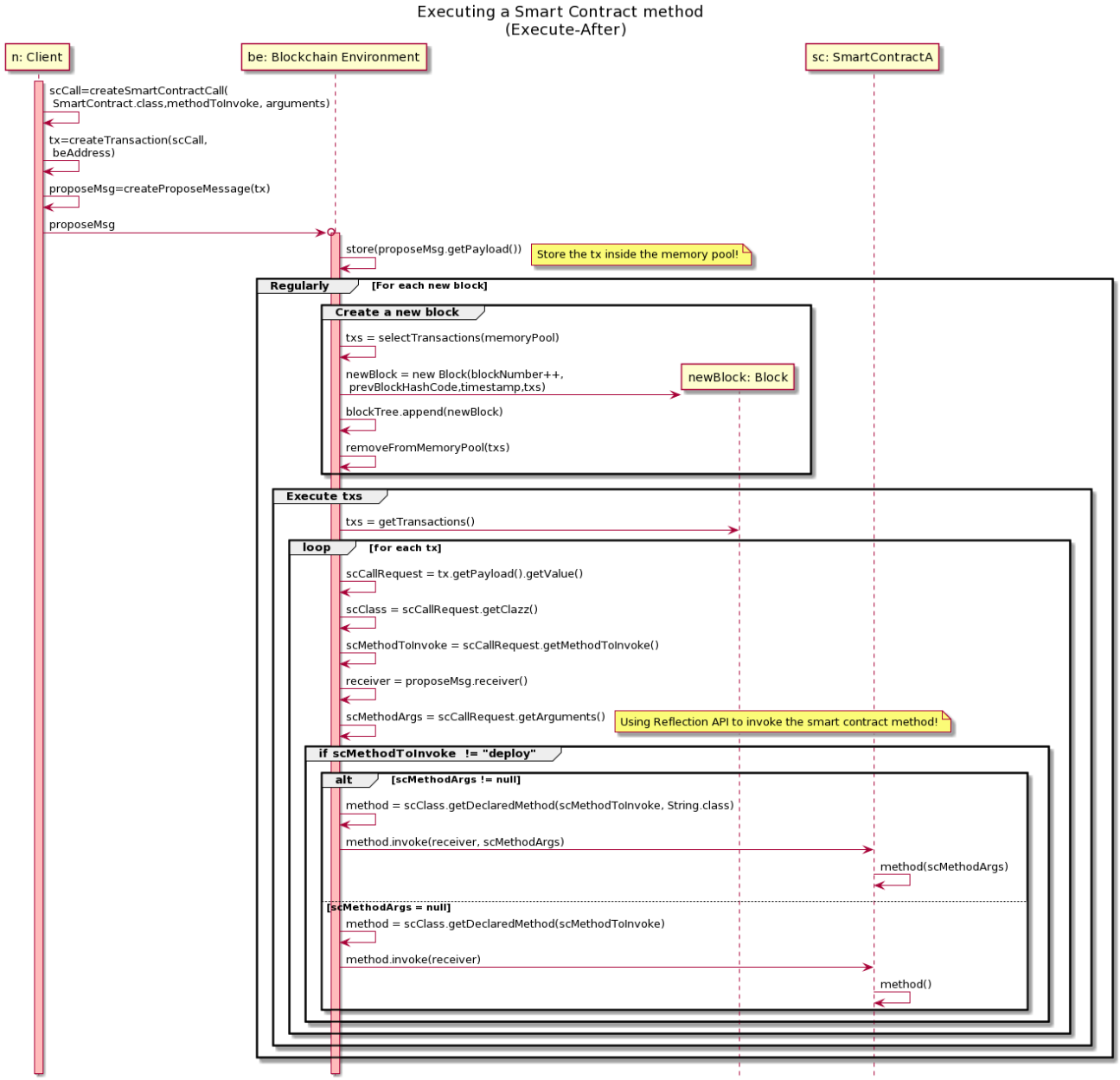


Figure 18: Smart Contract methods workflow : UML Diagram

6.3 Execute-First architecture (Hyperledger Fabric blockchain)

Considering the Hyperledger Fabric architecture, transaction should be executed instantly before creating blocks. This means that the transaction is first executed then validated. If it is valid, we add it to the memoryPool, if not, we do nothing. In our simulation, to represent the endorsement policy, we consider that a transaction is valid if it does not throw an exception and invalid if it does.

In our simulation, to imitate this behavior, two solutions are possible:

- Create a method that enables returning to the last state before the transaction is executed in the case of non validity.
- Create a copy of the same smart contract, execute the transaction considering the copy, validating the result. If the transaction is approved, we execute it considering the original smart contract and then add it to the memoryPool.

In our implementation, we take into account the second solution as demonstrated in Figure 19

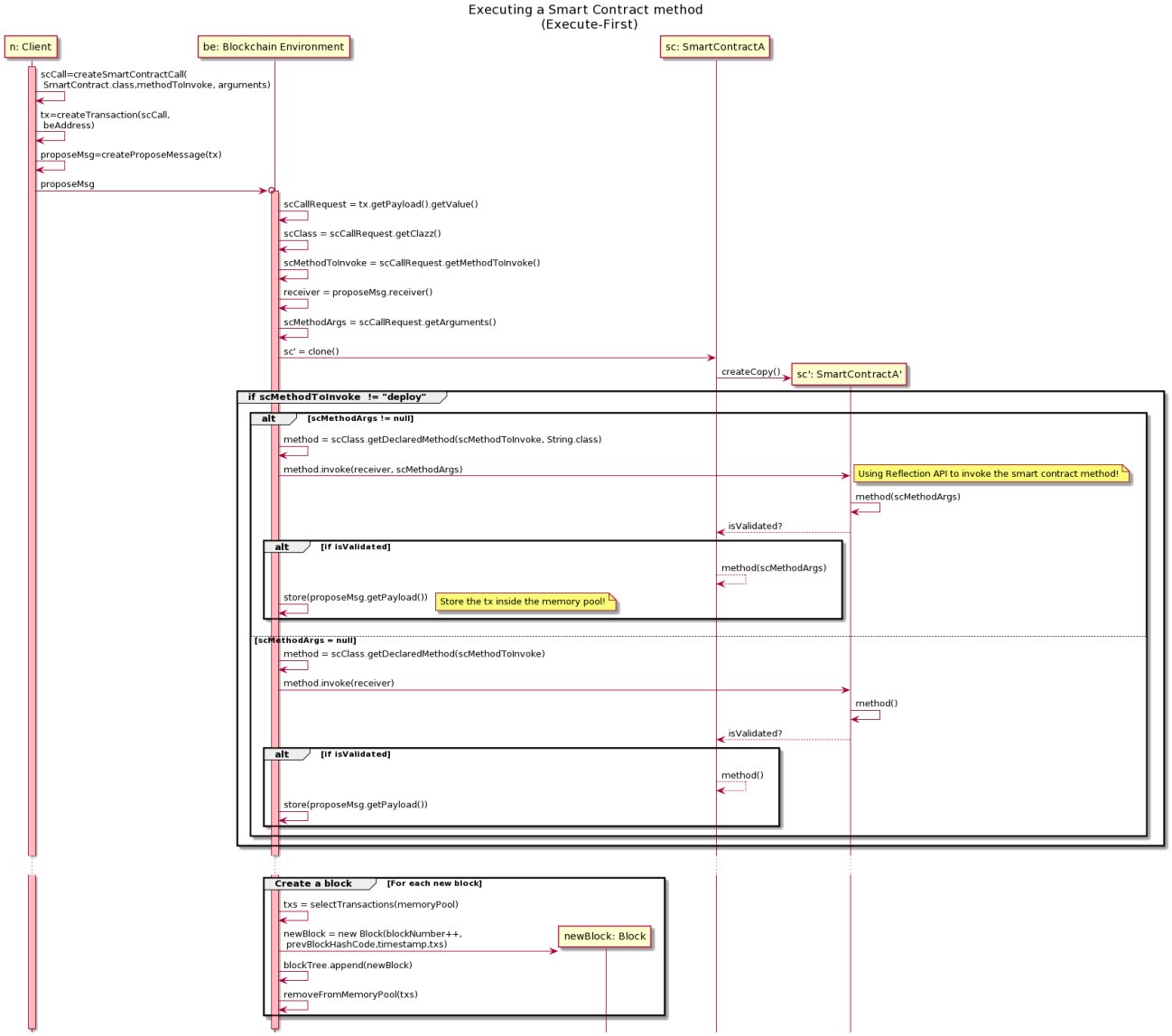


Figure 19: Smart contracts for Hyperledger Fabric blockchain

7 Use Case: An Industrial Prototype of Trusted Energy Performance Contracts

The purpose of the EPC [18] is to guarantee a lasting improvement in the energy efficiency of an existing building or group of buildings. It engages a contractor to design and deliver energy efficiency measures in order to improve energy efficiency by the means of reducing energy consumption. In this context, the objective is to evaluate the relevance of blockchain technologies using smart contracts to overcome the technical limitations mentioned.

The use case is implemented using the simulation model proposed considering the Execute-First architecture to simulate Hyperledger Fabric blockchain.

7.1 Simulations

In this study, we consider an existing agent-based simulator in the LICIA based on an Agent/Environment/Role model: Multi Agent eXperimenter (MAX), a multi-agents platform designed to power blockchain applications based on Madkit[40].

7.1.1 Multi Agent eXperimenter (MAX)

MAX uses two key concepts:

- *Agent*: autonomous entity with its own knowledge, preferences and behaviors.
- *Environment*: where agents are evolving.

Agents can observe and interact with the environment (Figure 20).

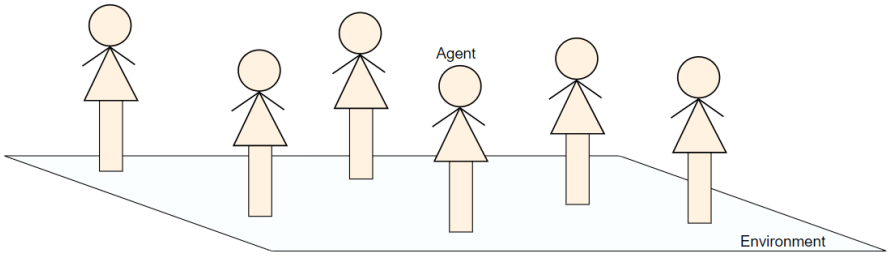


Figure 20: Representation of agents in an environment

To represent complex agent organizations, MAX is using the *Agent/Environment/Role* paradigm. Roles can be viewed as tags: they represents capabilities of an agent. Each agent can play different roles in an environment (Figure 21).

[40]MAdkit, `http://www.madkit.net/madkit/`

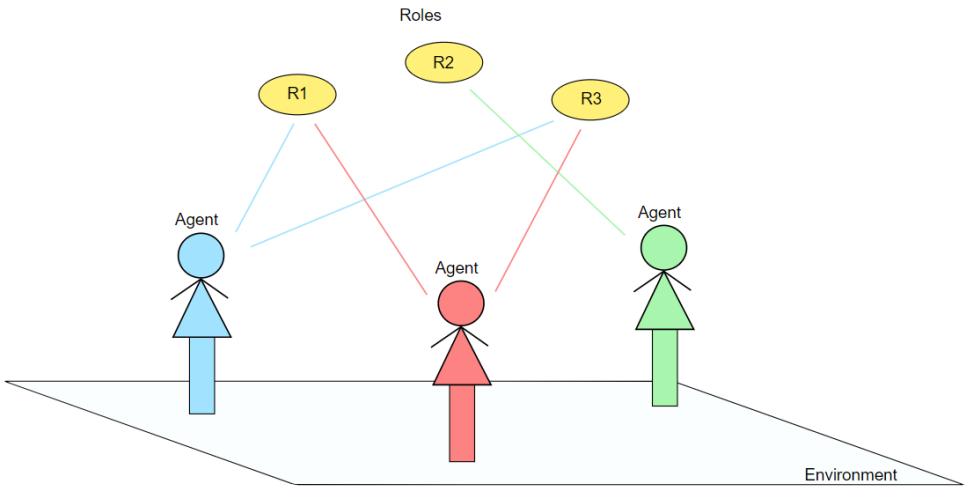


Figure 21: Representation of agents playing roles in an environment

The different agents defined by MAX are:

- *SimulatedAgent* is the base class of user-defined agents. It comes with basic functionalities and works by executing actions according to a *Plan* used to describe what an agent will do and when.
- *SimulatedEnvironment* is the base class to represent the Environment, which is also an Agent in MAX.
- *SchedulerAgent* is the agent responsible of scheduling and executing all actions in the simulation.

In such simulation, we use a *Context*, the agent's memory given a particular environment. The first time an agent joins an environment, the environment creates a *Context* for that agent. It then adds it to the list of contexts the agent already have. The notion of *Plan* used enables storing the actions of agents to run associated to a certain event type and schedule them.

7.1.2 Implementation

In this use case, the roles, actions and environment remain the same as implemented. What differs is the number of agents.

- We consider four smart contracts (Figure 22):

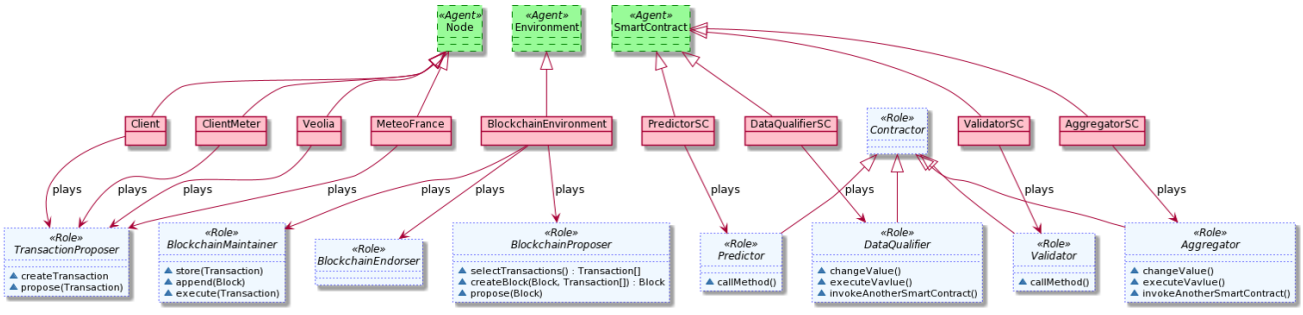


Figure 22: EPC diagram

– **DataQualifier** has the responsibility of qualifying the hourly data samples coming from *"authorized sources"*: *Veolia*, *Meteo France* and *Client* by calling the *Validator* smart contract. To do so, it verifies whether the data samples are inside the predefined range (temperature values between -30°and 60°, humidity values between 0% and 100%, pressure values between 850bar and 1060bar and consumption values higher than or equal to 0kW). If it is not the case it replaces the sample with the closest boundary value. Each hour, it proceeds a *"voting algorithm"* considering a degree of *"reliability"* as follows:

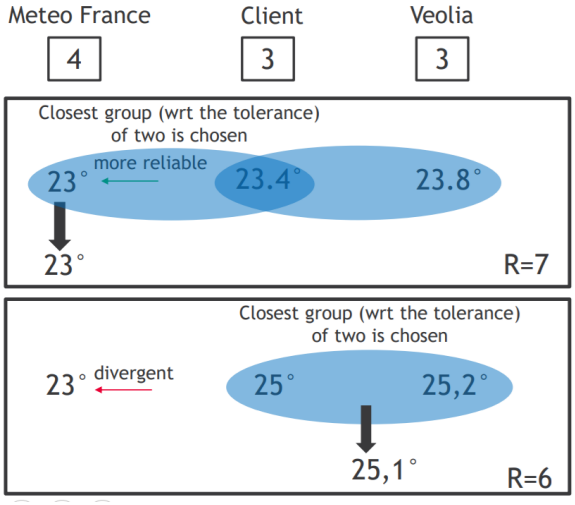


Figure 23: Voting Algorithm

  * Group the sources into subgroups of agreement between them wrt close tolerance.
  * Select the subgroup with the highest weight.
  * Compute the qualified value as the average of the source(s) with the greatest weight within the subgroup.
  * Compute the reliability as the sum of the votes of the chosen group.

  At end of the day, the *DataQualifier* contract passes all the hourly qualified samples to the *Predictor* contract to predict the daily saving.
– **Predictor** predicts the daily saving using the qualified daily wrt a certain predetermined algorithm provided. The output is then passed to the Aggregator smart contract.
– **Aggregator** aggregates daily saving as monthly savings by simply adding it to the the total saving. At end of the month, the contract calculates the final monthly saving in kW.
– **Validator** validates static variables of the client building and the monthly saving.

- We consider four nodes:
  – Client
  – Veolia
  – MeteoFrance
  – ClientMeter

7.2 The Daily Prediction Scenario:

The authorized sources send their hourly samples during a month, and a monthly saving is calculated by using the calculated daily predictions that are calculated by using qualified daily samples (Figure 24).

More specifically, the EPC scenario is as follows:

- Each hour, *Client*, *Veolia* and *Meteo France* provide their hourly samples to the *DataQualifier* smart contract. It then checks and validates the hourly samples.
- Each end of the day, *ClientMeter* provides the real consumption value to *DataQualifier* contract.
- Each end of the day, *DataQualifier* contract launch the daily qualification process by calling the predictor to calculate the daily saving by applying the *Veolia* algorithm to compute the saving and sends it to the *Aggregator* contract that will accumulate/add it to the total saving made.
- At the end of the month, the *Aggregator* contract passes the monthly saving to the *Validator* contract who will decide if a saving was made or not.

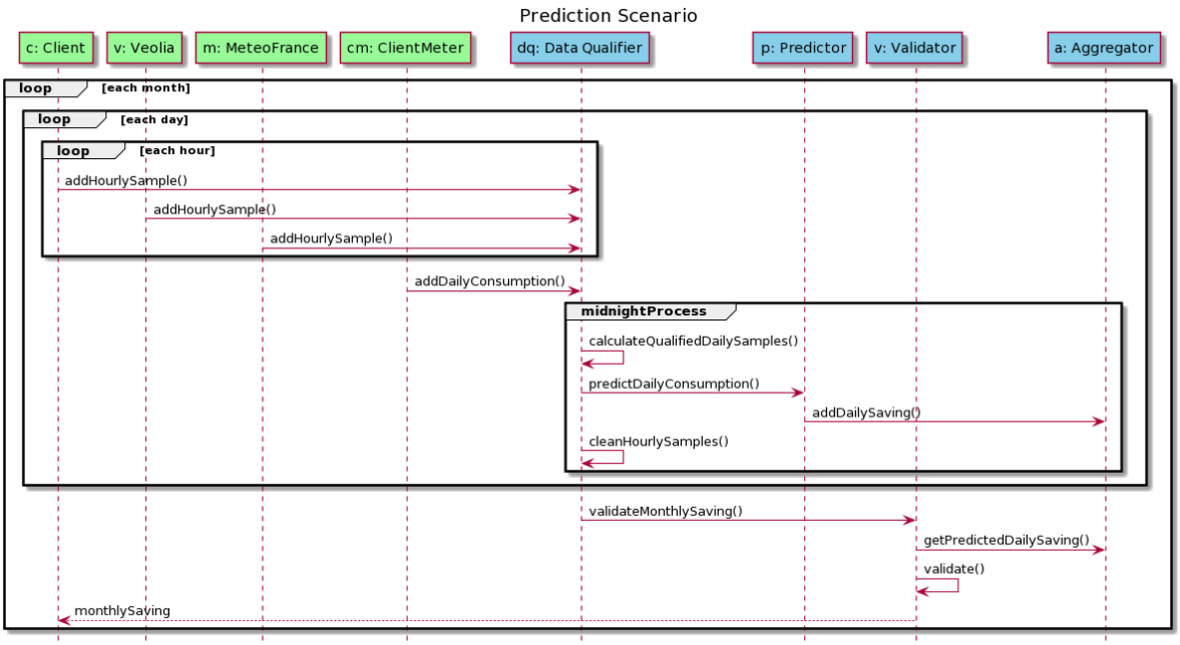


Figure 24: The Daily Scenario

7.3 Results

While launching the monthly scenario, we verified that contracts are correctly called and each action is executed at the fixed tick as shown in Figures 25, 26, 27.

```
// Create plan with RNode Role role
final Plan<Node> planToCreateSC = new Plan<>() {
    @Override
    public List<ActionActivator<Node>> getInitialPlan() {
        return Arrays.asList(
                new ActionActivator<>(ActivationScheduleFactory.ONE_SHOT_AT_STARTUP,
                        new ACTakeRole<>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(), RNode.class,
                                null)),

                // Deploy Data Qualifier contract at tick zero
                new ActionActivator<>(ActivationScheduleFactory.ONE_SHOT_AT_ONE,
                        new ACDeploySCEPC<Node>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(),
                                RNode.class, null)),

                // Read data samples for excel fill
                new ActionActivator<>(ActivationScheduleFactory.createOneTime(BigDecimal.valueOf(2)),
                        new ACCallSCEPC<Node>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(),
                                RNode.class, null, "readDataFileFor30Days", null)),
                // Compute the daily saving
                new ActionActivator<>(
                        ActivationScheduleFactory.createRepeatingFinitely(BigDecimal.valueOf(12 + 25),
                                BigDecimal.valueOf(24 * 31), BigDecimal.valueOf(24)),
                        new ACCallSCEPC<Node>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(),
                                RNode.class, null, "midnightProcess", null)),

                // At the end of the month, compute the monthly saving and validate it
                new ActionActivator<>(
                        ActivationScheduleFactory.createOneTime(BigDecimal.valueOf(13 + 24 * 31)),
                        new ACCallSCEPC<Node>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(),
                                RNode.class, null, "validateMonthlySaving", null))
                );
    }
};

timeOracle = new Node(planToCreateSC);
agents.add(timeOracle);
```

Figure 25: Time Plan

```
// Create plan with RNode Role role
Plan<Node> planForAuthorizedSources = new Plan<>() {
    @Override
    public List<ActionActivator<Node>> getInitialPlan() {
        return Arrays.asList(
                new ActionActivator<>(ActivationScheduleFactory.ONE_SHOT_AT_STARTUP,
                        new ACTakeRole<>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(), RNode.class,
                                null)),

                // Veolia, Meteo France and Client should provide samples each hour/tick
                new ActionActivator<>(
                        ActivationScheduleFactory.createRepeatingFinitely(BigDecimal.valueOf(12),
                                BigDecimal.valueOf(24 * 31), BigDecimal.ONE),
                        new ACAddSamples<Node>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(),
                                RNode.class, null, "addAuthorisedDataSample", "VEOLIA")));

    }
};

veolia = new Node(planForAuthorizedSources);
agents.add(veolia);

planForAuthorizedSources = new Plan<>() {
    @Override
    public List<ActionActivator<Node>> getInitialPlan() {
        return Arrays.asList(
                new ActionActivator<>(ActivationScheduleFactory.ONE_SHOT_AT_STARTUP,
                        new ACTakeRole<>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(), RNode.class,
                                null)),

                // Veolia, Meteo France and Client should provide samples each hour/tick
                new ActionActivator<>(
                        ActivationScheduleFactory.createRepeatingFinitely(BigDecimal.valueOf(12),
                                BigDecimal.valueOf(24 * 31), BigDecimal.ONE),
                        new ACAddSamples<Node>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(),
                                RNode.class, null, "addAuthorisedDataSample", "METEOFRANCE")));

    }
};

meteoFrance = new Node(planForAuthorizedSources);
agents.add(meteoFrance);
```

Figure 26: Veolia and Meteo France Plan

```
planForAuthorizedSources = new Plan<>() {
    @Override
    public List<ActionActivator<Node>> getInitialPlan() {
        return Arrays.asList(
                new ActionActivator<>(ActivationScheduleFactory.ONE_SHOT_AT_STARTUP,
                        new ACTakeRole<>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(), RNode.class,
                                null)),

                // Veolia, Meteo France and Client should provide samples each hour/tick
                new ActionActivator<>(
                        ActivationScheduleFactory.createRepeatingFinitely(BigDecimal.valueOf(12),
                                BigDecimal.valueOf(24 * 31), BigDecimal.ONE),
                        new ACAddSamples<Node>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(),
                                RNode.class, null, "addAuthorisedDataSample", "CLIENT")));
    }
};

client = new Node(planForAuthorizedSources);
agents.add(client);

final Plan<Node> planClientMeter = new Plan<>() {
    @Override
    public List<ActionActivator<Node>> getInitialPlan() {
        return Arrays.asList(
                new ActionActivator<>(ActivationScheduleFactory.ONE_SHOT_AT_STARTUP,
                        new ACTakeRole<>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(), RNode.class,
                                null)),
                // Client Meter should provide samples each day at exactly 23:59 (each 24 ticks)
                new ActionActivator<>(
                        ActivationScheduleFactory.createRepeatingFinitely(BigDecimal.valueOf(13 + 24),
                                BigDecimal.valueOf(24 * 31), BigDecimal.valueOf(24)),
                        new ACAddSamples<Node>(Organization.BLOCKCHAIN_ENVIRONMENT.toString(),
                                RNode.class, null, "addClientMeterConsumption", null)));
    }
};
clientMeter = new Node(planClientMeter);
agents.add(clientMeter);
```

Figure 27: Client and Client Meter Plan

When running the scenario considering the monthly data provided by Veolia, MeteoFrance, Client and ClientMeter, we have the same results with the two architectures.

The difference is when we give an inappropriate value to the DataQualifier (*i.e. a null value)*. In the case of the Execute-After architecture, an exception is thrown and the execution of the scenario is stopped.

```
        ... 18 more
java.lang.reflect.InvocationTargetException
        at jdk.internal.reflect.GeneratedMethodAccessor1.invoke(Unknown Source)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:564)
        at max.model.ledger.smartcontract.env.BlockchainEnvironment.callSmartContractMethod(BlockchainEnvironment.java:182)
        at max.model.ledger.smartcontract.action.ACExecuteTxAfter.execute(ACExecuteTxAfter.java:54)
        at max.core.scheduling.ActionActivator.execute(ActionActivator.java:110)
        at madkit.kernel.Activator.execute(Unknown Source)
        at madkit.kernel.Scheduler.executeAndLog(Unknown Source)
        at max.core.scheduling.SchedulerAgent.executeAndReschedule(SchedulerAgent.java:195)
        at max.core.scheduling.SchedulerAgent.doSimulationStep(SchedulerAgent.java:118)
        at madkit.kernel.Scheduler.live(Unknown Source)
        at madkit.kernel.Agent.living(Unknown Source)
        at madkit.kernel.AgentExecutor$2.run(Unknown Source)
        at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:515)
        at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
        at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
        at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:630)
        at java.base/java.lang.Thread.run(Thread.java:832)
Caused by: java.lang.NullPointerException: Cannot invoke "java.lang.Double.doubleValue()" because "null" is null
        at max.model.ledger.EPCSmartContract.action.DataQualifier.addAuthorisedDataSample(DataQualifier.java:82)
        ... 18 more
```

Figure 28: Exception thrown in the case of Execute-After architecture

While using the Execute-First architecture, the exception will also be thrown but once catched, the execution continues by neglecting the invalid transaction and thus not adding it to the memory pool.

```
Validate:
the actual saving is: -2055.7999999999997
```

Figure 29: Exception thrown in the case of Execute-First architecture

In our use case, detecting vulnerabilities is only managed by handling exceptions. This implementation allows us at first to simulate the smart contracts considering the different architectures but also to make a link with the state-of-the-art and to discover/test the smart contracts when exceptions are to consider.

8 Conclusion and Prospects

The main purpose of this internship is to provide insights into VV&T of smart contracts to assess the overall effectiveness of popular security code analysis solutions used to detect common vulnerabilities. The main motivation behind this work is to contribute to a more secure and trustworthy Hyperledger Fabric environment.

First, a comprehensive review is conducted on the available VV&T solutions for smart contracts.

The state-of-the-art outlines already exploited vulnerabilities for targeted blockchain and classifies them based on their severity and test level. It demonstrates that simulators are the more flexible solution to simulate smart contracts considering the three levels (DApp, Blockchain, Network) and the degree of control over the system parameters.

Hence, they are non existent simulators when given away the public ones. The only one available, as far as I found, is a private one developed by a startup. Moreover, Hyperledger smart contracts VV&T solutions are very few compared to other blockchains. To cover this point, we evolved an Agent/Environment/Role simulator to imitate smart contracts considering the DApp level for blockchain agnostic and then focus on the Hyperledger Fabric blockchain (considering the Execute-After and Execute-First architectures respectively).

To conclude this internship report and validate the model developed, the highlight that smart contracts are promising applications is demonstrated through the EPC use case developed by the mean of MAX, the agent-based simulator developed by the LICIA Lab.

To go even further, the project could be extended considering the blockchain and system levels to have more control and visibility of manipulating the blockchain parameters. In order to improve the smart contract process, setting up a simulation that detects a large kind of common vulnerabilities could be a better and significant approach to secure smart contracts before the deployment in the main networks.

Appendix

Vulnerabilities and attacks

Vulnerabilities	Description
Integer overflow/underflow (IOU)	Occurs when performing operations with value limitations, causing an overflow (resp. underflow)
Unchecked Send Bug	Referred to as "send instead of transfer. 'Transfer' automatically checks for the return value, whereas using 'send' you have to manually check for the return value, and throw an exception if the send fails. Not doing so, can lead to an attacker executing malicious code into the contract and draining the balance.
Mishandled Exceptions	Occurs because of the the interaction between contracts. If exceptions are not handled correctly and the transactions are reverted, the user will be unaware of ether lost
Gas costly pattern	Occurs when a smart contract execution consumes more gas unnecessarily
tx.origin	Concerns a global blockchain variable that refers to the address of the account sending the transaction. When it is used for authorization, contracts may be compromised by phishing attacks
Reentrency	Occurs when a smart contract function is being executed concurrently for another smart contracts through an outer call before the first function call was ended.
Front Running	Occurs when a user of the network react to an transaction before it is included in the next block.
Timestamp dependence (TD)	Occurs when a miner manipulate the block timestamp, thus changing the output of the contract to his own benefit.
Transaction ordering dependency (TOD)	Occurs when a miner decides an inconsistent transactions order with respect to the time of invocations, leading to malicious behaviors
Global State Variables	Variables are only global to a single peer and are not tracked on the ledger. Which can lead to a divergence between the peer global variable states.
Call to the unknown	Occurs when a function invocation or an ether transfer unexpectedly invokes the fallback function of the callee/recipient.

Table 5: Smart Contract vulnerabilities

Attacks	Description
Limiting progress attacks	Occurs when a malicious user slow down the entire protocol by delaying the broadcast of the first message that enables the update. Thus, no progress can be made.
DAO attack	Occurs when the malicious user asks the smart contract to provide the money back multiple times before the smart contract could update the token balance.
DoS attack	Implies leaving contracts dysfunctional for some time or even permanently. It occurs when the attacker overwhelms the system by large amounts of transactions that the blockchain is unable to handle or transmits bugs that exploit the blockcahin flaws .

Table 6: Smart Contract Attacks

Game of Life

In the late 1960s, the mathematician John Conway invented a virtual mathematical machine that performs actions based on certain rules [31]. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves. Each cell takes two states, live and dead. The cells' states are updated simultaneously and in discrete time.

The implementation was done following the agent based model (refer to the class diagram in Figure 30):

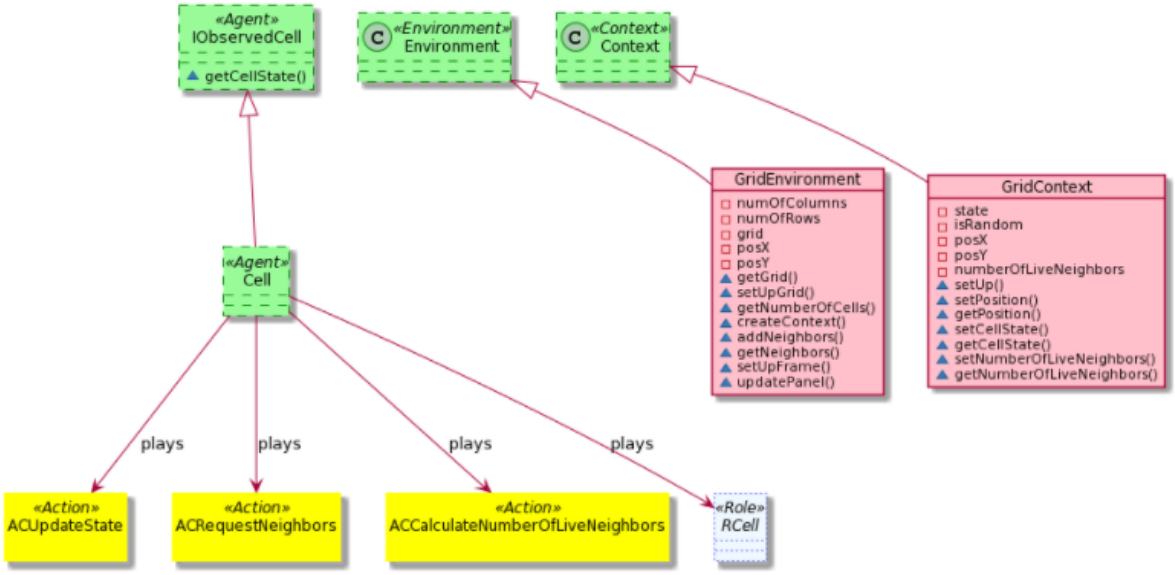


Figure 30: Game of life Class Diagram

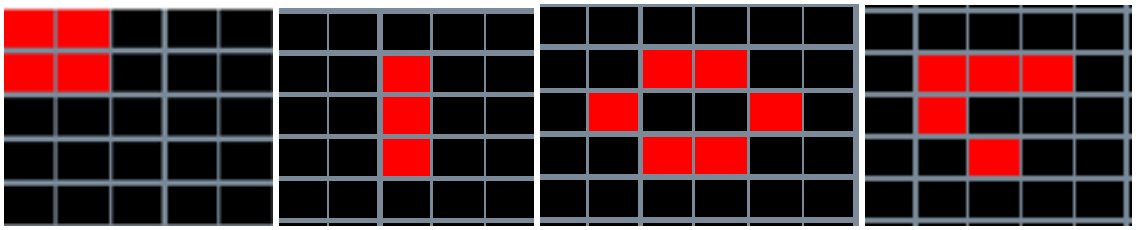
Entities, Roles and Behaviors:

The model is composed of a *GridEnvironment*, *GridContext*, *IObservedCells* and *Cells*.

- *GridEnvironment* constructs environment for the game of life, register the *Cell* agents and set up the visualization of the grid. It also enables counting the number of live neighbors of each agent cell.
- *GridContext* enables the cell to update its state according to three defined actions:
  - *ACRequestNeighbors*: is executed at tick one to register the neighbors of each cell in its context.
  - *ACCalculateNumberOfLiveNeighbours*: count the number of live neighbors considering the state of the IObservedCells.
  - *ACUpdateState*: change the cell state according to four rules:
    - Any living cell with fewer than two live neighbors dies (underpopulation).
    - Any living cell with four or more neighbors die (overpopulation).
    - Any cell with exactly two neighbors remain in its state.
    - Any cell with exactly three neighbors become/remain alive.
- *Cell* agent plays *RCell* role. Each agent in the grid has a *RCell* role.
- *IObservedCells* is a cell interface that enables other agents to observe the cell state without accessing to its context.
- *GameOfLifeExperimenter* plays the RExperimenter role. It initializes a grid of size 50x50 and enables the agent cells with *RCell* role and *ACRequestNeighbors*, *ACCalculateNumberOfLiveNeighbors* and *ACUpdateState* actions. It is responsible for running a simulation and initializing cell behaviors randomly and with a certain density. The life cycle of this experimentation is as follows:
  - First, the scheduler requests the neighbors of each cell agent,
  - Then the scheduler calculate the number of live neighbors by observing their state,

- After, the scheduler update the cell agent state according the initial state and to states progress.
- These steps are repeated until the end of the simulation.

After running multiple simulations, we found some recurring shapes/patterns: blocks, gliders, beehives and blinkers (see Figure 31).



(a) (b) (c) (d)

Figure 31: Recurrent shapes/patterns

References

[1] *A Generic Multi-Agent Organizational Model for Blockchain Systems*, "submitted". 2021.

[2] Mouhamad Almakhour, Layth Sliman, A. Samhat, and A. Mellouk. *"Verification of smart contracts: A survey"*. *Pervasive Mob. Comput.*, 2020.

[3] Elli Androulaki, Artem Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, S. Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. Cocco, and Jason Yellick. *"Hyperledger fabric: a distributed operating system for permissioned blockchains"*. *Proc of the 13th EuroSys Conf*, 2018.

[4] Monika Di Angelo and G. Salzer. *"A survey of tools for analyzing Ethereum smart contracts"*. *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, 2019.

[5] G. Angeris, H.-T. Kao, R. Chiang, C. Noyes, and T. Chitra. *"An Analysis of Uniswap Markets"*. *ERN: Other Microeconomics: General Equilibrium & Disequilibrium Models of Financial Markets*, 2019.

[6] Pandurang Kamat Siddhartha Chatterjee Arati Baliga, I Subhod. *"Performance Evaluation of the Quorum Blockchain Platform"*. 2018.

[7] O. Balci. *"Validation, verification, and testing techniques throughout the life cycle of a simulation study."* in proc. of the 26th conf. on winter simulation, wsc'94, pages 215–220, san diego, ca, usa. society for computer simulation international. 1994.

[8] L. Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. *"Vandal: A scalable security analysis framework for smart contracts"*. *ArXiv*, 2018.

[9] Carlyle J. Grigg I. Brown, R.G. and M. Hearn. *"Corda: an introduction."*. 2016.

[10] Ilene Burnstein. *Practical Software Testing*. Springer, 2003.

[11] T. Li X. Luo G. Gu Y. Zhang C. Ting, R. Cao and Z. Liao. *"SODA: A Generic Online Detection Framework for Smart Contracts"* ,in network and distributed system security symposium. 2020.

[12] Radu State Christof Ferreira Torres, Julian Schütte. *"Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts."*, proceedings of the 34th annual computer security applications conference. 2018.

[13] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, B. Ooi, and K. Tan. Blockbench: A framework for analyzing private blockchains. *Proc of the 2017 ACM Int Conf on Management of Data*, 2017.

[14] J. Eberhardt and S. Tai. *"Zokrates: scalable privacy-preserving off-chain computations."* in ieee international conference on internet of things and ieee green computing and communications and ieee cyber, physical and social computingand ieee smart data. 2018.

[15] Grieco G. Feist, J. and A. Groce. *"Slither: a static analysis frame- work for smart contracts."* in proceedings of the 2nd international work- shop on emerging trends in software engineering for blockchain. 2019.

[16] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From agents to organizations: An organizational view of multi-agent systems. volume 2935, pages 214–230, 01 2003.

[17] LM Goodman. Tezos—a self-amending crypto-ledger white paper. 2014.

[18] Ö. Gürcan, M. Agenis-Nevers, Y. Batany, M. Elmtiri, F. Le Fevre, and S. Tucci-Piergiovanni. *"An Industrial Prototype of Trusted Energy Performance Contracts Using Blockchain Technologies"*. In *2018 IEEE 20th Int Conf on HPC and Comm, IEEE 16th Int Conf on Smart City, IEEE 4th Int Conf on Data Sci and Syst*, 2018.

[19] D. Harz and W. Knottenbelt. *"Towards Safer Smart Contracts: A Survey of Languages and Verification Methods"*. *ArXiv*, 2018.

[20] Y. Huang, Yiyang Bian, Renpu Li, J. Zhao, and Peizhong Shi. *"Smart Contract Security: A Software Lifecycle Perspective"*. *IEEE Access*, 2019.

[21] Olivier Gutknecht Jacques Ferber and Fabien Michel. *"From Agents to Organizations: An Organizational View of Multi-agent Systems "*. 2003.

[22] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. *"ZEUS: Analyzing Safety of Smart Contracts"*. In *NDSS*, 2018.

[23] Ence Zhou Bingfeng Pi Kazuhiro Yamashita, Yoshihide Nomura and Sun Jun. *"Potential Risks of Hyperledger Fabric Smart Contracts"*, in ieee int workshop on blockchain oriented software engineering (iwbose). 2019.

[24] A. Kosba, Andrew K. Miller, E. Shi, Zikai Wen, and Charalampos Papamanthou. *"Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts"*. *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[25] Chao Liu, H. Liu, Zhao Cao, Z. Chen, Bangdao Chen, and A. W. Roscoe. *"ReGuard: Finding Reentrancy Bugs in Smart Contracts"*. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, 2018.

[26] Loi Luu, D. Chu, Hrishi Olickel, P. Saxena, and Aquinas Hobor. *"Making Smart Contracts Smarter"*. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[27] Lodovica Marchesi, M. Marchesi, and R. Tonelli. *"ABCDE - Agile Block Chain Dapp Engineering"*. *ArXiv*, 2019.

[28] Monika di Angelo Marco Bareis and Gernot Salzer. *"Functional Differences of Neo and Ethereum as Smart Contract Platforms."*published in 2nd international congress on blockchain and applications. 2020.

[29] David Mazières Graydon Hoare Nicolas Barry Eli Gafni† Jonathan Jove Rafał Malinowsky Marta Lokhava, Giuliano Losa and Jed McCaleb. *"Fast and secure global payments with Stellar."* proceedings of the 27th acm symposium on operating systems principles. 2019.

[30] A. Mavridou and A. Laszka. *"Designing secure Ethereum smart contracts: a finite state machine based approach."* in financial cryptography and data security - 22nd international conference. 2018.

[31] Laszka A. Stachtiari E. Mavridou, A. and A. Dubey. *"Verisolid: correct-by-design smart contracts for Ethereum."* in financial cryptography and data security - 23rd international conference. 2019.

[32] A. Mense and M. Flatscher. *"Security Vulnerabilities in Ethereum Smart Contracts"*. *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, 2018.

[33] Ghassan O. Karame Lucas Davi Michael Rodler, Wenting Li. *"Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks"*., in proc of 26th annual network distributed system security symp (ndss). 2018.

[34] M. Bartoletti N. Atzei and T. Cimoli. *"A survey of attacks on ethereum smart contracts"*.proceedings of the 6th international conference on principles of security and trust. 2017.

[35] Satoshi Nakamoto. *"Bitcoin : A Peer-to-Peer Electronic Cash System"*. 2008.

[36] Zeinab Nehai and François Bobot. *"Deductive Proof of Ethereum Smart Contracts Using Why3"*. *ArXiv*, abs/1904.11281, 2019.

[37] Danny Weyns · Andrea Omicini · James Odell. *"Environment as a first class abstraction in multiagent systems"*. 2007.

[38] Quang-Trung Ta Meihui Zhang Gang Chen Beng Chin Ooi Pingcheng Ruan, Dumitrel Loghin. *"A Transactional Perspective on Execute-order-validate Blockchains."*. 2020.

[39] Munish Saini Rajdeep Kaur, Kuljit Kaur Chahal. *"Understanding community participation and engagement in open source software Projects: A systematic mapping study."*, journal of king saud university - computer and information sciences,. 2020.

[40] Christian Sillaber and Bernhard Waltl. *"Life Cycle of Smart Contracts in Blockchain Ecosystems."*. *Datenschutz und Datensicherheit - DuD*, 41:497–500, 2017.

[41] F. Spoto. *"A java framework for smart contracts."* in financial cryptography and data security - fc 2019 international workshops. 2019.

[42] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. *"Securify: Practical Security Analysis of Smart Contracts"*. *Proc of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[43] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[44] Z. Yang and H. Lei. *"Fether: an extensible definitional interpreter for smart-contract verifications in coq."* ieee access 7, 37770–37791. 2019.