DM

DEMO Model-Driven Automatic Smart Contract Generation using Hyperledger Fabric

MASTER DISSERTATION

Leonardo Tadeu Nunes Abreu
MASTER IN INFORMATICS ENGINEERING

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

September | 2025

DEMO Model-Driven Automatic Smart Contract Generation using Hyperledger Fabric

MASTER DISSERTATION

Leonardo Tadeu Nunes Abreu
MASTER IN INFORMATICS ENGINEERING

SUPERVISOR
David Sardinha Andrade Aveiro

Faculty of Exact Sciences and Engineering

Master of Science Degree in Informatics Engineering

DEMO Model-Driven Automatic Smart Contract Generation using Hyperledger Fabric

Leonardo Tadeu Nunes Abreu

Supervised by:

David Sardinha Andrade de Aveiro

Constitution of the Public Examination Jury:

Karolina Baras (Assistant Professor), Presidente

Fábio Rúben Silva Mendonça (Assistant Professor), Vogal

David Sardinha Andrade de Aveiro (Assistant Professor), Vogal

Friday 31st October, 2025

Summary

Esta dissertação de mestrado estuda a geração automática de smart contracts através de modelos organizacionais, com foco na área da logística e cenários de economia circular. Esta investigação emergiu do contexto do projeto MiCoLEC, que visa a criação de um micro-hub colaborativo para transporte de encomendas e logística reversa.

Este trabalho adota o DEMO (Design and Engineering Methodology for Organizations) de modo a especificar formalmente as regras organizacionais e aplicá-las na plataforma Hyperledger Fabric, utilizando a linguagem de programação Go. Foi definida uma extensão da gramática EBNF para representar as regras de ação DEMO, com um método de mapeamento que permite a sua transformação em chaincode executável. Para melhorar a manutenção, foi proposto um processo estruturado de geração de ficheiros.

A abordagem foi validada através da implementação de smart contracts que suportam as principais operações da plataforma MiCoLEC, incluindo leilões, entregas de encomendas, participação de estafetas, envolvimento de clientes na logística reversa e mecanismos de inventivo para a economia circular. Os resultados foram desenvolvidos corretamente e executáveis no Hyperledger Fabric e são capazes de suportar os requisitos funcionais da plataforma.

Esta investigação contribui para reduzir a barreira ao desenvolvimento de smart contracts, ao estabelecer uma ligação entre modelação empresarial e a implementação em blockchain. Fornecendo assim, uma base para soluções low-code que promovem a interoperabilidade, transparência e confiança em colaborações interorganizacionais.

Keywords: Enterprise engineering · DEMO Action Model · Blockchain · Hyperledger Fabric · Smart Contracts · Logistics industry

Abstract

This dissertation investigates the automatic generation of smart contracts from organizational models, with a focus on the logistics domain and circular economy scenarios. The research is carried out in the context of the MiCoLEC project, which aims to establish collaborative micro-hubs for parcel delivery and reverse logistics.

The work adopts the DEMO (Design and Engineering Methodology for Organizations) methodology to formally specify organizational rules and applies them to the Hyperledger Fabric blockchain platform using the Go programming language. An extention of EBNF grammar for representing DEMO Action Rules is defined, together with a transpilation method that enables their transformation into executable chaincode. To improve maintainability, a structured file generation process is also proposed.

The approach was validated through the implementation of smart contracts that support the main operations of the MiCoLEC platform, including parcel delivery auctions, courier participation, customer engagement in reverse logistics, and incentive mechanisms for circular economy operators. The results show that the generated contracts were correct and executable within Hyperledger Fabric, and capable of supporting the functional requirements of the platform.

This research contributes to lowering the barriers of smart contract development by linking enterprise modeling with blockchain implementation. It provides a foundation for low-code solutions that promote interoperability, transparency, and trust in interorganizational collaborations.

Keywords: Enterprise engineering · DEMO Action Model · Blockchain · Hyperledger Fabric · Smart Contracts · Logistics industry

Agradecimentos

Esta dissertação só foi possível com ao apoio de pessoas fundamentais na minha vida, tanto a nível pessoal como profissional. A todos, deixo o meu sincero agradecimento.

Em primeiro lugar, ao meu orientador, Prof. David Sardinha Andrade de Aveiro, pela orientação, disponibilidade e incentivo constantes. Foi o verdadeiro impulsionador desta dissertação e espero que muitos outros tenham a oportunidade de beneficiar da sua orientação. Muito obrigado por todo o apoio dado nesta etapa da minha vida.

À minha namorada, Lia Carolina Góis Abreu, com quem partilho 12 anos de relação à data da entrega desta dissertação. Pelo carinho, paciência e apoio incondicional, não poderia pedir melhor companheira de vida.

Aos meus pais, irmãos e familiares mais próximos, pela motivação e por estarem sempre presentes nos momentos de maior fragilidade.

Aos meus amigos e antigos colegas de trabalho, pelo incentivo e força transmitida, em especial ao Tiago Xavier e ao André Gouveia.

Por fim, mas provavelmente os mais importantes os meus dois cunhados. São duas crianças excecionais com características totalmente diferentes. O Salvador Matias Góis Berenguer, uma criança incrível e com um futuro brilhante, que anima qualquer pessoa com o seu riso contagiante. A Mia Valentina Góis Berenguer, a verdadeira definição de princesinha, uma menina com muito afeto e carinho, sempre disponível a dar o seu abraço para reconfortar quem está mais vulnerável. Um muito obrigado por estarem presentes na minha vida.

A todos, o meu obrigado.

Table of Contents

List of Figures

List of Tables

List of Acronyms

BFT Byzantine Fault Tolerance

CEOp Circular Economy Operators

DASCG DEMO Automatic Smart Contract Generation

DEMO Design and Engineering Methodology for Organizations

DPoS Delegated Proof-of-Stake

EE Enterprise Engineering

LOps Logistic Operators

MiCoLEC Micro-hubs Colaborativos para a Economia Circular

NFTs Non-Fungible Tokens

P2P Peer to Peer

PA Platform Administrator

PBFT Practical Byzantine Fault Tolerance

PoS Proof of Stake

PoW Proof of Work

1 Introduction

This chapter introduces the research of this dissertation, including its motivation, the problem addressed, and the objectives pursued. It begins with a discussion of the context in which the work is situated, followed by the definition of the problem and its relation to the MiCoLEC project the basis of this dissertation. The chapter then sets out the objectives of the dissertation, the adopted methodology, and concludes with the outline of the document.

This thesis emerged in the context of the project Micro-hubs Colaborativos para a Economia Circular (MiCoLEC), which aims to establish a collaborative logistics micro-hub. Collaborative micro-hubs are a new logistical concept in which a group of delivery companies collaborate among themselves by sharing means and resources of delivery of shipments in a network of common logistical centers installed in strategic areas of urban centers, essentially with a view to operating costs reduction, service quality improvement and coverage of more geographic areas without large initial investments. On the other hand, this same concept makes it possible to reduce urban traffic, with all the resulting advantages, namely the reduction of air pollution, the number of road accidents and improvement of the quality of life within urban areas [1]. Despite the enormous advantages of collaborative micro-hubs, their implementation is a complex challenge, whose success lies in the implementation of solutions that solve the problem of trust between different transport operators in the exchange of information and execution of shared operations between them.

The circular economy aspect brings together a set of reuse, recycling and sustainable end-of-life processes for the massive amount of products that the consumer economy produces daily, presenting itself today as the most viable and promising way to reach the necessary levels of environmental sustainability that the planet urgently needs [2]. The advantage of the circular economy in relation to other ecologically sustainable philosophies of life is based on the fact that the processes it promotes can be introduced without significant changes in the lifestyle of modern societies. However, the effective implementation of circular economy processes depends on solutions to a set of challenges, among which the need to create economic models to encourage the continued participation of final consumers in circular economy processes, and the availability of supply of reverse logistics services (from the consumer) with adequate levels of cost and quality of service.

This project developed a practical solution that addressed these two problems, through the application of blockchain technology (distributed ledger technologies) for the implementation of a

digital solution that solved the collaboration and trust issues at the base of the limitations of implementing collaborative micro-hubs. The project implementation is able to record all transactions of the micro-hub in a verifiable, permanent, and transparent way for all interested stakeholders.

Furthermore, the project introduced a digital token (cryptocurrency), called “bitcircle,” whose management was also codified in the blockchain and which was attributed to final consumers and logistical operators as an incentive to participate in circular economy processes. Both the allocation of digital tokens to end consumers and operators and the management of transactions by the exchange of services between transport companies were carried out using smart contracts, which were also registered in a transparent and verifiable way for all stakeholders on the blockchain.

In MiCoLEC project, we developed smart contracts in Go for Hyperledger Fabric to ensure that rules are enforced transparently and cannot be tampered with. We also implemented a Gin API (application programming interface) that is responsible to interact with the smart contracts deployed in the Hyperledger Fabric, acting like a bridge between smart contract methods and the developed Laravel backend application.

In MiCoLEC project we also explore the viability of the DEMO (esign and Engineering Methodology for Organizations) Model-Driven automatic smart contract generation, we create a a transpiling method between DEMO action rules and Chaincode Go, that resulted in a conference publication of the paper "DEMO Models Based Automatic Smart Contract Generation: a case in Logistics using Hyperledger" [3]. We also created a proof of concept of the automatic smart contract generation which resulted in another conference publication "Towards Automatic Smart Contract Generation from DEMO Models, a case in Logistics using Hyperledger" [4]

1.1 Motivation

Currently, global markets face increasing demands for transparency in production of goods. This demand comes from both end-users and regulators, who seek detailed information about the entire production process, from the sourcing of raw materials to manufacturing methods and the logistics involved in the transportation of the final product. To reach this level of demand, robust and reliable tracking systems are needed.

Centralized systems are widely adopted in the supply chain. However, they suffer from a lack of transparency and trust among participants. On the other hand, decentralized systems are gen-

erally more secure, transparent and reliable. They are not controlled by a single entity or central authority. These systems ensure data protection, reduce inconsistencies and eliminate single points of failure. They also help mitigate fraud, ensure regulatory compliance, and improve transparency in the logistics sector [5]. To address these issues, blockchain technology emerges as an alternative. Blockchain is one of the most promising technologies today. It is a decentralized database where transactions are stored in cryptographically linked blocks, forming an immutable chain commonly referred to as a ledger [6, 7]. In contrast to traditional centralized systems, blockchain presents a decentralized system operating within a peer-to-peer (P2P) network. In this network, all participants (nodes) possess a synchronized copy of the ledger. Each node assumes the responsibility of validating and recording blocks onto the ledger. [7].

To guarantee data integrity, blockchain technology relies on consensus protocols that ensure all nodes agree on the current state of the ledger. Each block is validated by the participants before being added to the ledger.

There are several consensus protocols, such as "Proof of Work" (PoW) or "Proof of Stake" (PoS). This document will later explore the most commonly used protocols in more detail. Consensus protocols allow the network to maintain agreement on data states without the need for a central authority [8].

Blockchain technology has evolved through three main phases. Initially known as “Blockchain 1.0” was exclusively used for cryptocurrencies, particularly Bitcoin. In this phase, blockchain systems securely enabled financial transactions without intermediaries.

Following the first phase, “Blockchain 2.0” introduced smart contracts, enabling automated transactions. These are programs that run on top of the blockchain system, following pre-determined rules established by interested parties [7].

The idea of smart contracts was firstly introduced by Nick Szabo [9] in 1990, the author defines a smart contract as a self-executing agreement when pre-determined conditions are met. Nick Szabo’s primary objective is to minimize the reliance on intermediaries. By ensuring that all parties involved adhere to their respective obligations within the agreement framework, Nick Szabo aims to achieve a safe and efficient system.

The introduction of smart contracts in blockchain has a major impact on the adoption of blockchain in other fields, before smart contracts the use of blockchain was limited for cryptocurrency, after smart contract the blockchain gained interest in distributed finances, supply chains, NFTs (Non-Fungible Tokens) and others.

Blockchain 3.0 focuses on speed and scalability, creating new opportunities when combined with emerging technologies like AI and IoT. This evolution enables new blockchain applications in healthcare, education, and logistics. [6, 7].

1.2 Problem

The development of smart contracts remains a demanding task that requires proficiency in specific programming languages and familiarity with blockchain infrastructures. This technical barrier constitutes a limitation for projects such as MiCoLEC, where the effective operation of a collaborative logistics micro-hub depends on transparent and verifiable execution of interorganizational rules.

In practice, the specification of these rules often exist at the organizational level, described in terms of business processes and agreements between logistics operators, couriers, circular economy operators, and consumers. Translating such specifications into executable smart contracts normally involves manual programming. This introduces risks of inconsistency, delays in implementation, and difficulties in maintenance, particularly when rules evolve. For MiCoLEC, these limitations directly affect the capacity to enforce trust and coordination among participants, as the validity of parcel auctions, delivery assignments, and token-based incentives relies on precise execution of shared rules.

Another problem arises from the predominant reliance on text-based notations for contract design. These notations are difficult to interpret by non-technical stakeholders, creating a gap between organizational requirements and their blockchain implementation. In MiCoLEC, where multiple actors with different technical backgrounds must collaborate, this gap can reduce the adoption of the platform and the confidence of participants in the system.

Finally, existing frameworks for smart contract development provide limited support for traceability between organizational models and generated code. This reduces transparency and complicates audits, which are critical for a platform such as MiCoLEC that aims to ensure fairness and accountability in parcel delivery and circular economy transactions.

These issues highlight the need for an approach that bridges organizational modeling and blockchain implementation, reducing technical barriers, improving accessibility, and ensuring a direct correspondence between business rules and smart contract execution.

1.2.1 Research Questions

The challenges outlined in the previous highlight the need for a systematic approach that bridges organizational modeling and blockchain implementation. In this context, the present dissertation seeks to address specific research questions that guide the investigation and define the scope of the proposed solution.

Research Questions:

- RQ1: How can correct and executable smart contracts be automatically generated from DEMO Action Rules within a blockchain platform?
- RQ2: How can the proposed approach reduce technical barriers in smart contract development within the MiCoLEC context?

1.3 Objetives

The main objective of this research is to reduce the barriers associated with the creation of smart contracts by introducing a model-driven approach based on DEMO and applied to Hyperledger Fabric, by reverse engineering of implemented smart contracts and devising of generic rules allowing automatic generation.

In the context of MiCoLEC, this objective translates into three concrete goals:

- **Support parcel delivery auctions and negotiations**: Define a grammar and a transpilation process capable of transforming DEMO Action Rules into Chaincode, enabling logistics operators to offer and bid parcel deliveries. This ensures that the bidding process and its outcomes are transparently recorded on the blockchain.
- **Enable participation of couriers and end customers**: Generate smart contracts that govern the assignment of deliveries to couriers and the engagement of end customers in reverse logistics. These contracts provide verifiable mechanisms for execution and rewards, ensuring that participants can interact under predefined and auditable conditions.

- **Incorporate incentives for circular economy operators**: Implement contracts that encode the attribution of rewards to circular economy operators for receiving parcels of interest. This objective contributes to reinforcing recycling and reuse processes by codifying incentive mechanisms directly in the blockchain.
- **Reduce barriers to adoption through model-driven generation**: Provide a visual and structured path from organizational models to executable smart contracts, reducing the need for specialized programming knowledge. This facilitates adoption of the MiCoLEC platform by diverse stakeholders and ensures that operational rules remain transparent and consistent.

By achieving these objectives, the research aims to contribute both to the MiCoLEC project and to the broader challenge of making blockchain-based collaboration more accessible, reliable, and aligned with enterprise engineering principles.

1.4 Document Outline

This document is organized into six chapters.

In Chapter 1, we introduce the context of the research, describing the motivation for adopting blockchain technology in collaborative logistics and circular economy scenarios. It introduces the problem addressed, defines the objectives of the work, and situates the research within the scope of the MiCoLEC project.

In Chapter 2, Literature Review examines the theoretical foundations and related work relevant to this research. It reviews the main concepts of blockchain, smart contracts, and their application in logistics, followed by a discussion of existing approaches for automatic smart contract generation. The review identifies gaps in the literature that the present work aims to address.

Chapter 3, we explore the context and requirements, introduce the MiCoLEC project in detail and analyze the applicability of blockchain to its operational needs. It defines the beneral objectives and functional requirements of the MiCoLEC platform, supported by user profiles and user stories, and concludes with the specification of the system architecture and specific technical requirements that were implemented in the context of this dissertation.

In Chapter 4 we explore the implemented solution. It details the APIs used to interface with the blockchain, the design and implementation of smart contracts in Hyperledger Fabric, and

the proposed approach for automatic smart contract generation from DEMO models, including examples and file structures.

In Chapter 5, Results and Discussion we present the results obtained from the implementation and evaluate them based on the objectives defined in Chapter 1. The chapter discusses the correctness of the generated smart contracts, their execution within the MiCoLEC context, and the maintainability of the generated file structure, while also comparing the approach with related work and identifying limitations.

To conclude on Chapter 6 Conclusion, we summarize the main contributions of the dissertation, highlight the limitations of the current work, and suggest directions for future research, particularly the extension of the grammar extensions created in the context of this project, the application of the method to other blockchain platforms, and the systematic evaluation of performance and scalability.

2 Literature Review

This chapter reviews the theoretical and technical foundations relevant to the research. It first presents the concepts of blockchain (2.1) and smart contracts (2.2), followed by their application in logistics (2.3) and circular economy contexts. The chapter then examines approaches to model-driven development and automatic generation of smart contracts (2.4). The review concludes (2.5) by identifying gaps in the literature that frame the contribution of this work.

2.1 Blockchain

Blockchain is a decentralized, distributed system on a P2P network. Transactions are stored in blocks, which are validated by network nodes and appended to the chain, creating an immutable and secure ledger. Nodes on the P2P network are independent and trustless with each other [7].

To fully grasp the concept of blockchain, it is essential to delve into its core features: decentralization, persistence, anonymity, and auditability [6].

- **Decentralization:** traditional centralized systems regards on a central authority to maintain the data consistency, this approach can lead us to possible problems: performance, cost and unfair treatment for some participants. In decentralized systems, no third party are needed, and data consistency is maintained by using consensus algorithms.

- **Persistency:** Given the decentralized nature of blockchain, all transactions undergo validation by the network. Consequently, invalid transactions are rejected by honest nodes, commonly referred to as miners. In these systems, it is challenging to delete or revert transactions once they have been appended to the ledger.

- **Anonymity:** Users interact with the blockchain using a generated address, usually called a wallet address, which does not reveal the identity of the user. Blockchain allow us to track what is done but not who does what, because user privacy is maintained and user interact with the blockchain using the wallet address.

- **Auditability:** Since all transaction are transparent to the network, nodes can easily see and verify all transaction present on the network. Auditability give us a public full history and easy to track tampered data.

2.1.1 Block Structure

A block in blockchain is composed of a header and body, the block header contains all the general information about the block, while the block body contains the list of transactions, Figure 1 illustrates the block structure. [6].

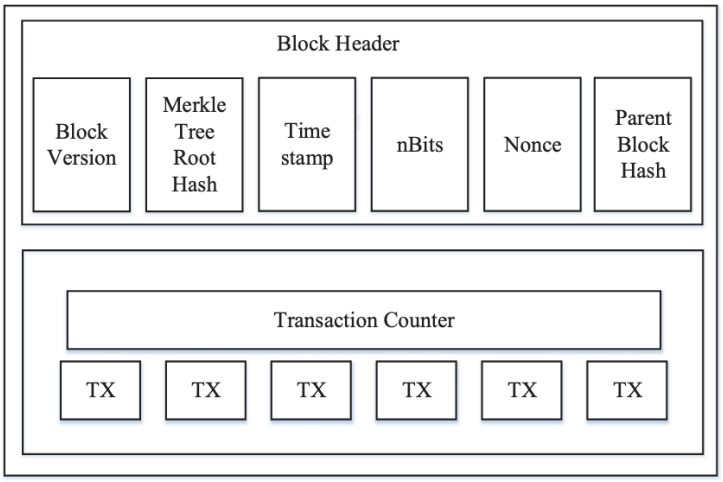


Figure. 1: Block structure [6]

– **Block Header:**

- **Block version:** represents the set of rules that are used to validate the block.
- **Merkle tree root hash:** this component represents the hash value of all transaction that are present in the block.
- **Timestamp:** This field is used to keep track of the current time.
- **nBits:** This field is a value that defines the level of difficulty to finding a valid hash for the block.
- **Nonce:** This value is a number that starts with 0 and is increased in order to find a valid hash value.
- **Parent Block Hash:** a 256-bit hash that contains the hash value of the previous block.

– **Block Body:**

- **Transaction Counter:** It holds the number of transactions present on the block.

- **List of transactions:** list of transactions present in the block.

2.1.2 Ledger

Previously, it was observed that each block has a hash of the previous block, this is used to create the chain of blocks, illustrated on Figure 2, also called a ledger, this ledger creates an historical and immutable data transactions timeline. We can affirm that it is immutable because if anyone tries to change a block that is already on the ledger, the hash of that block will be different and will not match the one that is supposed to be in the next block, and it is easily spot and deny that change for the rest of the network [6].

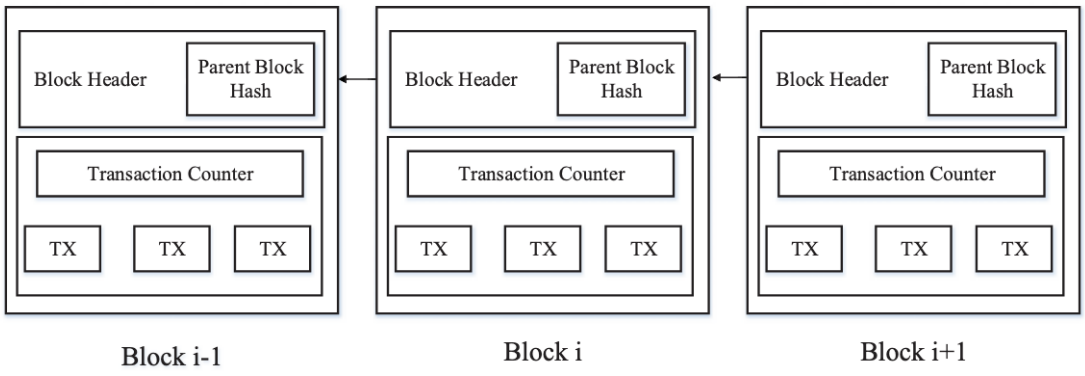


Figure. 2: An example of blockchain which consists of a continuous sequence of blocks. [6]

2.1.3 Peer-to-peer Network (P2P)

A P2P network is a decentralized network architecture, where participants, known as peers, have equal capabilities and responsibilities. When discussing blockchain technology, the entities that participate in the network are referred to as nodes instead of peers, which are computers that possess the same level of access and capabilities within the network.

Centralized database systems have some drawbacks in security and lack of transparency. In terms of security, since this system has a single point of control, this also means a single point of failure, that can be used by malicious users in order to compromise the entire system. When referring to lack of transparency, this leads to a lack of trust among participants, specially in case where data integrity is critical, and it is controlled only by a single authority [10].

Different from traditional system where client-server models with a central authority managing communication, in P2P networks we have all nodes having this responsibility. This decentralization nature of P2P network makes these systems ideal for applications, like file sharing, distributed computing and as mention previously, blockchain technology [11, 12].

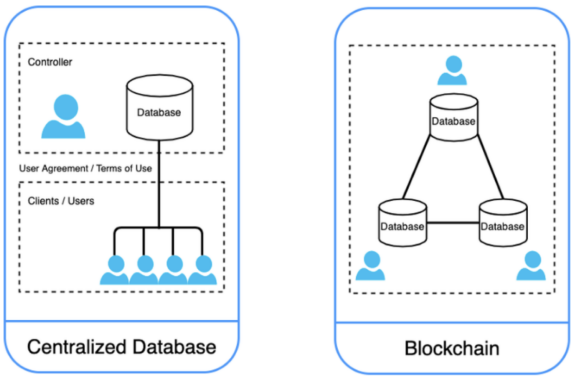


Figure. 3: Centralized database solutions and Blockchain [10]

2.1.4 Hash and Cryptography

In this section, we will briefly delve into the role of hash and cryptography within the blockchain system. However, we will keep things simple and not dive too deep into the details, since that is not really what we are here to explore.

In blockchain systems, hashing ensures data integrity by generating a unique "fingerprint" for each block. Hash function are unidirectional functions that take an input and generate a fixed-size output, the hash. Even the smallest change in the input will generate a different hash value, being this a crucial for data integrity because if any data on a block is changed it will generate an entirely different hash value, and the other nodes will spot easily that that block was suffered a change and will not accept that change. Different blockchain systems use different hash functions, for example: Bitcoin and Hyperledger uses SHA-256, while Litecoin and Dogecoin uses SCRYPT as hash function [13].

In terms of cryptography, blockchain makes use of public-key cryptography, widely known as asymmetric cryptography. This system uses a pair of keys: public and private key. The public key can be shared with anyone, while the private key must be kept in secret by the owner. Users will sign transactions using their private keys, and the network will verify their validity using the

corresponding public key. This process ensures that only authorized user can commit transactions in their name [14].

A good example of asymmetric cryptography was given by Zheng et al. [6], they used an example where one person wants to send a message to another one. And the that message passes by the two phases of asymmetric cryptography, signing and verification.

But we think it is easier to understand with an example where things are traded. Imagine the example where we have a user who wants to sell a car but wants a payment using a cryptocurrency. The buyer must send the right amount of money to the seller, so the buyer will sign the transaction with the amount agreed upon with the seller. This represents phase one signing. In the second phase, all the nodes must validate the transaction using the buyer's public key. If the transaction was not changed by anyone, it will be accepted and added to the block. If any user tries to change the data when decoding using the public key, it will be easier to detect if the data has been tampered.

Hashing and cryptography are essential components of the blockchain and work together to ensure security and integrity of the blockchain [13, 14].

2.1.5 How blockchain works

To elucidate how blockchain works, we will use Figure 4 used by Sarmah et al. [15] this figure illustrates the typical blockchain flow when a user initiates a transaction.

Firstly, the user sends the transaction, the system will add it to a block, this block will contain all the components referred previously, and it is important to mention that the block will contain in it's a header a timestamp and a hash of the previous block, also called parent block, this is important because these properties are important to secure the date integrity on the ledger.

Once the block is ready to be sent to the network, the user has to broadcast it to every single node participating in the network. In the next phase, the consensus algorithm is used to validate transactions and plays a crucial role in maintaining the safety and efficiency of the blockchain, we will look at in detail in the next subsection. After validated by the network, the block is ready to be added to the ledger.

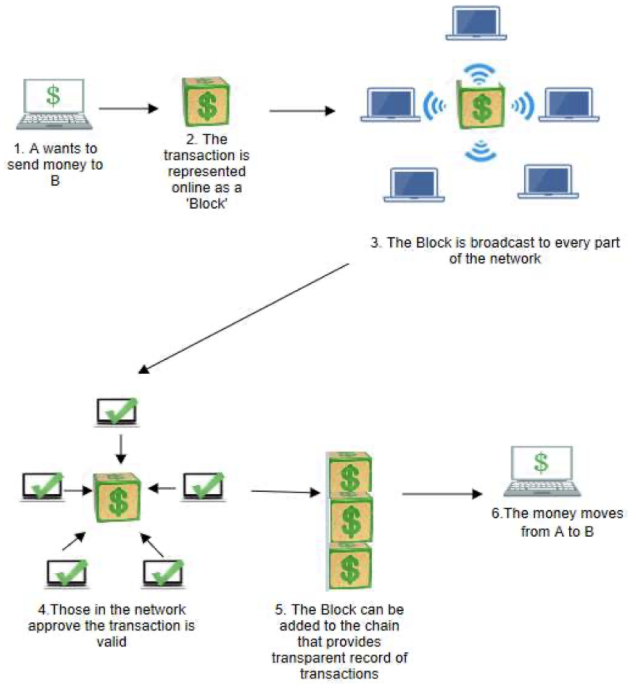


Figure. 4: How Blockchain works [15]

2.1.6 Consensus Algorithms

Blockchain Consensus algorithms can be defined as a set of rules through which node in the blockchain network as to reach an agreement regarding the current state of the distributed ledger. These algorithms ensure that all nodes agree and validate the chronological order in which transactions are added to the blockchain ledger, this is a way to maintain data integrity and consistency across the blockchain network. Being one of the most fundamental technologies inside the blockchain world. In summary, consensus algorithms are mechanisms that bring security to the distributed network like blockchain, allowing all nodes to reach a consensus of the current state of the data stored on the ledger [16, 17].

The functionality of blockchain consensus algorithms is frequently discussed within the context of Byzantine Fault Tolerance (BFT), which deals with the challenge of reaching agreements within a distributed network where some participants are unreliable or malicious. These algorithms are designed to provide a BFT, this means that they have the capability of ensuring that the network will operate correctly and reach agreement, even if a certain proportion of participants act dishonestly.

This ability to tolerate faults, particularly those caused by malicious behavior, is a critical aspect of the security and reliability that consensus algorithms provide to blockchain networks. This ability to tolerate faults, particularly those caused by malicious behavior, is a critical aspect of the security and reliability that consensus algorithms provide to blockchain networks [18].

Among multiple consensus mechanisms, we chose the four most used: Proof of Work (PoW), Proof of Stake (PoS), Delegated Proof of Stake (DPoS) and Practical Byzantine Fault Tolerance (PBFT). Each of these algorithms has distinct features that make them more appropriate for specific use cases [19].

PoW is widely used to secure and validate transactions in a blockchain network without depending on a central authority. It is used in cryptocurrency, being first popularized by Bitcoin. The main goal of PoW is to achieve agreement among nodes, ensuring that integrity and immutability of the ledger. This consensus algorithm requires computational effort, this property makes it resistance to attacks and ensure security within the blockchain network [20, 21].

Now, let us delve into the practical functionality of PoW in the context of blockchain technology.

1. **Transaction Verification** - When a new transaction is broadcast to the network, nodes verify its validity. This involves checking the transaction's syntax, ensuring the sender has the necessary funds, and confirming that the transaction follows the network's rules.

2. **Block Formation** - Multiple transactions are grouped together into a block. As seen before, each block includes a header and a body. The header contains metadata such as the block number, a hash of the previous block, and a timestamp. The body contains the transactions themselves

3. **Hash Function and Target** - Miners start competing to find a hash that meets a specific target. The most blockchain systems, including Bitcoin, use the hash function SHA-256. The target is a number that the hash must be less or equal to. This process necessitates miners to execute multiple iterations of the hash function, employing diverse inputs, until the predetermined target is achieved [22].

4. **Mining Process** - In order to solve the complex mathematical puzzle, miners use powerful pieces of hardware, these mathematical problems involve the search for a nonce (random

number) that produces the hash that meets the target value difficulty. Since it requires computational power, we found the first and biggest draw back of PoW, high usage of energy [21].

5. **Block Validation and Addition to Blockchain** - When a miner successfully finds a valid hash, the block is broadcast to the network. All nodes present on the network has the responsibility to validate the block hash, if it is a valid block, then this block will be appended to the ledger. This process ensures that all nodes have the same version of the ledger and maintains the integrity of the blockchain [23].

6. **Incentivization** - Miners are incentivized to participate in the PoW process through block rewards and transaction fees. The first miner that can solve the cryptographic problem is rewarded with cryptocurrency and fees from the transactions included in the block. This reward system incentives the participants to secure the blockchain network. [24]

The PoW consensus algorithm brings several advantages that contribute essentially to security and decentralization of blockchain networks.

In terms of security, PoW is resistant to attacks from dishonest miners, in order to have a significative attack the attackers most control more than 50% of the network mining power, this in a network like Bitcoin makes nearly impossible to achieve, that is why this type of consensus algorithm are secure, having computing power is more rewarding to play the fair game and win the transaction fees and rewards, and ensure that the blockchain remains secure and functional [21].

Despite the strengths of PoW, this consensus algorithm is facing some challenges from energy consumption, to centralization of mining power and scalability issues

In terms of energy consumption, PoW requires substantial computational power to solve complex mathematical puzzles, leading to high-energy consumption. This energy-intensive process results in a significant carbon footprint, contributing to environmental concerns [20, 25].

Regarding scalability issues, PoW is facing challenges due to the limitation of transaction processing rate. Due to the computer intensive nature of PoW, the block generation time is relatively slow, leading to congestion and high transaction fees during peak times, making it less suitable for applications requiring high throughput [26].

PoW is designed to be decentralized, the high cost of mining hardware and energy can lead to centralization because of this kind of cost cannot be supported by single individuals, this leads

to large mining pools or entities that combine resources, which makes this networks centralized dominating the process power of the network [25].

PoW is widely used in blockchain system's like Bitcoin and Litecoin, for example, but some systems are changing from PoW to other consensus algorithms. The main cause of this change is due to the need of specialized hardware and high electricity costs, make these systems economically out of range for smaller participants [25]. For example, Ethereum in 2022 change the PoW to PoS, to break the economic barrier and allow new participants in the consensus algorithm.

PoS has emerged in the consensus algorithm world as a suitable alternative to the energy-intensive PoW. Instead of relying on computational power to solve a cryptographic problem, PoS selects validators to create the new blocks based on the number of tokens they hold and the number of tokens they are willing to "stake" as collateral [27].

In functionality perspective, the block generation on PoS is more simple than PoW, in this type of consensus algorithm, a group of validators is selected, to generate a new block, by the amount of cryptocurrency they hold and the amount they are willing to bet. This stake acts like a security measure, to incentive honest behaviors because if someone takes a malicious action it could result in loss of their staked assets [28].

PoS solves the problems of energy efficiency and the need for high computer power, but has some challenges that PoW already had but brings new ones, like centralization and wealth concentration, security vulnerabilities.

Let's start with scalability, PoS being more efficient than PoW, this scalability remains a challenge, particularly in a maintaining network performance and security as the number of participants grows [29].

In PoS also face problems with centralization, in PoW we have mining pools as a problem, in PoS have to deal with staking pools, since the participants that hold a higher stake have higher probability of being selected as validators, these groups of individuals can manipulate the network. This leads to a wealth concentration, leading to the problem "richer get richer", which creates barriers for new participants to have an active role in the consensus protocol [30].

Another risk would be that a 51% attack would be required to successfully control a network to accept fraudulent transactions. This is impractical as it requires the attacker to have a higher

investment of ownership of 51% of the finite number of coins in the cryptocurrency ecosystem. In large blockchain networks like Ethereum, 51% of the stake is almost impossible to reach because it means having approximately 72 billion US Dollars ($72,059,812,500.00) [28], but for smaller blockchain networks this can lead to security problems.

The PoS consensus algorithm is used in some well-known blockchains like Ethereum and Cardano.

DPoS

The DPoS is a variant of the PoS, this variant aims for better scalability and throughput problems identified on PoS algorithm. The functionality of DPoS is similar to the PoS, but on DPoS the number of consensus participants is reduced. In DPoS the token holders vote to elect a small number of participants, who are responsible for creating and validating blocks on behalf of the whole network. [31]

Since DPoS has fewer nodes on the creating and validating process, DPoS has the capability to process a high number of transactions per second, DPoS has high throughput and scalability, making it suitable for large-scale application.

In terms of energy efficiency, unlike PoW which uses significant computer power, DPoS relies on voting and stake delegation, making less block creators and validators being more energy-efficient.

DPoS has a governance flexibility, that allows governance to vote out poor performers and open a new voting pool for their place on consensus algorithm.

When looking at the weaknesses of DPoS, we pinpoint two main problems: centralization of consensus and security. In terms of centralization, since we have few delegated nodes participating in the consensus process, the creation and validation of new blocks are concentrated on the users who have higher stake. This leads to the second problem, security is a major problem of DPoS because we have fewer validating nodes, an attacker need only to compromise a few nodes to attack the network.

We can see that DPoS it's a trade between performance and security when compared to the previews explored consensus algorithms (PoW and PoS) [32].

PBFT consensus algorithm, like the other presented before, is designed to achieve agreement among distributed nodes in a network, even in the presence of faulty or malicious nodes.

PBFT works in 4 phases [33]:

1. pre-prepare: Primary node broadcast the proposing new block.
2. prepare: All network nodes receive the pre-prepare message from the primary node and verify, after validation, nodes broadcast their prepare message through the network.
3. commit: Nodes wait for a quorum of prepare messages before broadcasting the commit message, indicating the readiness to finalize the block.
4. decision: Once a node receives a quorum of commit message, a new block will be added to the ledger.

This algorithm looks simple, but it's efficient and secure, PBFT can tolerate up to (n-1)/3 faulty nodes in a network of n nodes.

In terms of performance, PBFT has a lower latency when compared to PoW due to its deterministic finality, but it can be affected by network size and communication. The same occurs to the throughput that is generally high for small networks, but when the network increases, throughput is penalized due to the communication complexity. These problems also affect the scalability of the blockchain network [33]. This consensus is most used in consortium blockchain like Hyperledger Fabric.

The consensus algorithms are one of the fields on blockchain systems that are being explored by research, since they have high impact on the adoption of blockchain systems in key characteristics such as scalability, security and governance [35].

2.1.7 Blockchain Network Types: Public, Private, and Consortium

Blockchain systems have evolved into three primary categories: public, private and consortium blockchain. Each type of blockchain was designed to fulfill specific purposes and has unique strengths and limitations, in terms of scalability, security and decentralization.

Public blockchains are decentralized networks that allow anyone to participate in the consensus process, these types of blockchains are the most widely known, for example Bitcoin and Ethereum.

Public blockchain operates on a decentralized network, which means not having a single entity controlling the system. This type of decentralization is achieved using consensus algorithms, dis-

Features	PoW	PoS	DPoS	PBFT
Types of blockchain	Permissionless	Permissioned & Permissionless	Permissioned & Permissionless	Permissioned
Decentralization structure	Strong	Strong	Strong	Weak
Electing miners based on	Solving difficulty hash	Stake owned	Stake owned	Mathematical operation
Reward	Yes	Yes	Yes	Yes
Speed of verification	Greater than 100 sec	Less than 100 sec	Less than 100 sec	Less than 10 sec
Speed block creation	Low	High	High	High
Consumption of energy	High	Less than PoW	Low	Moderate
Scalability	Strong	Strong	Strong	Weak
Fees of transaction	For all miners	For all miners	For all miners	No
Properties of distributed consensus	Probabilistic	Probabilistic	Probabilistic	Deterministic
Crash fault tolerate	50%	50%	50%	33%

Table 1: Comparison of consensus algorithms [34]

cussed in the previous section, PoW or PoS [35, 36]. Since anyone can participate in the consensus process, we have numerous validators, which make the network highly resistant to attacks [37]. Public blockchain is open to anyone, what makes transparency one of is strengths, since all transaction are recorded publicly, ensuring transparency and accountability [36, 38].

In terms of limitations, public blockchains often struggle with scalability due to the high computational requirements of their consensus algorithms [37], having the example of Bitcoin PoW limits the network to approximately 7 transactions per second [38, 39]. The energy required to maintain the network, specially when using PoW, has raised environmental concerns [38].

Another problem faced by public blockchains due to the decentralized nature can lead to regulatory challenges, as they operate across jurisdictions with varying legal frameworks [40]. In Figure5 we can see that some countries like Pakistan, Iran, Ecuador etc. are prohibiting the use of this type of cryptocurrency.

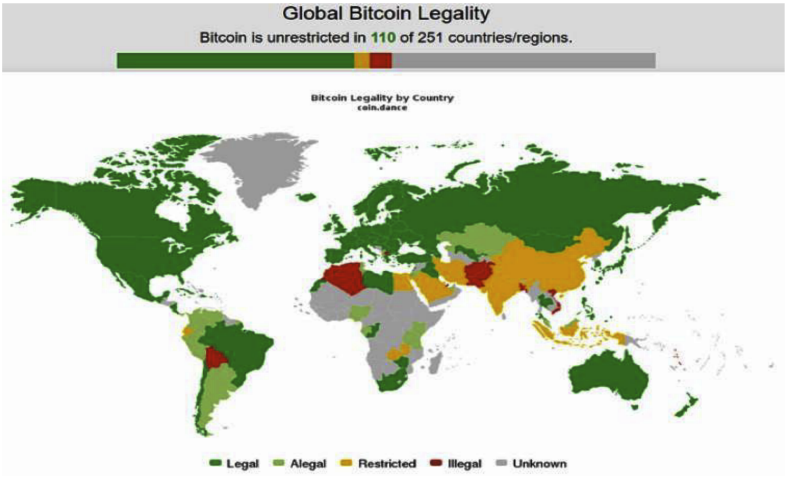


Figure. 5: Global Bitcoin Legality [40]

Private blockchains are permissioned networks, this means that only authorized nodes can join and participate in the consensus process. This type of blockchain is mainly used within organizations or between trusted partners [41].

Private blockchains generally have higher transaction throughput when compared to public blockchain, since the number of participants is considerable smaller the consensus is reached faster, enhancing high efficiency, faster transaction which leads to better scalability. This type of blockchain also offers better control over who can access the network and what data is shared, making it a suitable option for enterprise applications where privacy is a must [41].

Due to the fewer participants, private blockchains are way more centralized than public blockchain, which can lead to single point of failure. This can also lead to security problems, since we have a smaller number of nodes this makes it easier for a malicious node to manipulate the ledger special if the majority of nodes are compromised [41].

Consortium blockchains are a hybrid of public and private blockchains, where can a group of trusted nodes can validate transactions. These are often used in collaborative environments, where multiple organizations need to work together [42].

Consortium blockchain offer a balance between decentralization and centralization, they are in the middle, more decentralized than private blockchain but more centralized than public blockchains. This allows consortium blockchains to get higher transaction throughput than public and maintain a certain level of decentralization. In terms of security, the limited number of trusted nodes

reduces the risk of malicious actors compromising the network, making it more secure than private blockchains [42].

Consortium blockchain, by the other hand, has a complex governance since the decisions must be made collectively by the participating organizations, leading to slower decision-making process [42]. Due to this, consortium are less widely used than public or private blockchains.

Comparative Analysis

The comparative analysis, presented in table 2 evaluates public, private, and consortium blockchains across the dimensions of scalability, security, and decentralization.

Blockchain Type	**Scalability**	**Security**	**Decentralization**
Public	Low	High	High
Private	High	Medium	Low
Consortium	Medium	High	Medium

Table 2: Comparison between different types of blockchain.

In conclusion, public, private, and consortium blockchains each possess unique strengths and limitations. Public blockchains excel in decentralization and security, but have some scalability challenges. Private blockchains, on the other hand, prioritize scalability but compromise decentralization, potentially leading to security vulnerabilities. Consensus blockchains is a balance between scalability, security, and decentralization, making them well-suited for collaborative environments.

2.2 Smart Contracts

Blockchain technology has revolutionized the way transactions are conducted, and smart contracts have emerged as a pivotal component of this decentralized ecosystem. Nick Szabo in the 1990s [9] defined a smart contract as a computerized transaction protocol that executes the terms of a contract, in simpler terms, a smart contract is code that embodies contractual clauses and executes outcomes without the need for manual intervention.

Smart contracts are self-executing scripts embedded on a blockchain, enabling the automation of predefined rules and actions [43]. These contracts are computer programs that encode the terms of an agreement, and when the conditions are met, it will automatically execute. The core concept of smart contracts lies in their ability to facilitate trustless transactions, ensuring transparency, immutability, and security without relying on centralized authorities [44].

Smart contracts have become integral to developing decentralized applications (DApps), this evolution brings a flexibility and security, opening new opportunities for blockchain opportunities in new fields like finances, supply chain management, etc... [44].

A smart contract, as said before, is a self-executing contract where terms are written in code, in its essence is a program running on top of a blockchain system, and the objective of the program is to execute, control or document action based on conditions without the need of intermediaries.

In order to understand how smart contracts work in practice, we will use an example of a vending machine. Let's imagine that the smart contract is a vending machine, when you interact with a vending machine putting money (input), and then choose a product (action), the vending machine without intermediaries checks the input and the action validating if you have enough money for the product you request, if the condition is meet then the machine releases the product (output), otherwise do nothing. The vending machine works without the need of a third party, smart contracts work the same way but in a distributed network, where we have multiple nodes validating the same input and action, using the same contract, and the validation of the output is agreed using a consensus algorithm.

2.2.1 Challenges and Limitation on Smart Contract Development

Smart contracts can be seen as simple programs running on top of blockchain systems, but their development faces several challenges in terms of scalability, security and programming languages limitations [45, 46].

Smart contracts have high requirements for code security because they normally handle with sensitive digital assets like cryptocurrency or sensitive information. When smart contracts are deployed, the code cannot be modified and any security flaw can lead to irreversible financial losses [45].

The tools available for smart contract development are still basic, developers lack powerful interactive debuggers and tools for verifying code correctness, making it difficulty to ensure the quality and security of the code developed.

Most smart contracts are developed using Solidity (programming language used on Ethereum smart contracts), which is still evolving and has several limitations. These include a lack of general-

purpose libraries, poor error logging support, and inconvenient ways to call external functions [45], these problems are transversal to other used languages like Chaincode used on Hyperledger Fabric.

Another problem is the limitation of online resources and community support [45], this makes it difficult for developers, especially beginners, to find guidance and best practices.

Code visibility is another problem of smart contract development because the smart contract is visible to all participants, this can be problematic for companies that need to keep their business logic confidential, and it also makes it easier for attackers to find and exploit vulnerabilities [46]. Code updatability is another problem pointed on Kannagierse et al. [46], once deployed, smart contracts on Ethereum cannot be easily updated due to their tamper-resistant nature. This makes it challenging to correct flaws or add new features without deploying a new contract version.

Developers are also facing problems related to randomness, since generating secure random values is difficult because all nodes must produce the same random values to maintain determinism. This is a challenge across Ethereum, Hyperledger Fabric and other platforms [46].

In Hyperledger Fabric, smart contracts written in Go can face issues with concurrency, leading to nondeterministic behavior if not properly synchronized, this appends because Go is designed for parallel execution and supports concurrent execution using goroutines, which are functions that run concurrently with other functions. [46].

In response to these challenges, researchers are actively pursuing the automatic generation of smart contracts. This emerging field aims to streamline the development process of these contracts, enhancing their accessibility, efficiency, and security. Various approaches have been proposed, each leveraging different technologies and methodologies, to address the challenges associated with smart contract creation. These approaches can be broadly categorized into methods utilizing Large Language models (LLMs), annotation-assisted techniques, configuration-based systems, named entity recognition, and model-based strategies. We will delve into this matter further in the subsequent section of this document.

2.3 Blockchain on Logistics

Modern supply chain logistics are facing problems related to complexity, opacity and mistrust among parties. Products pass through many hands and processes from raw materials to the final

consumer, managing and maintaining all phases from raw materials until the final product without losing auditability and trust is a challenge that supply chains are facing nowadays [47].

Blockchain presents a potential solution to supply chain challenges. It enhances transparency and trust within supply chains by recording transactions on a tamper-proof shared ledger. This ensures that all participants have access to a single source of truth.

Blockchain combined with the strengths of smart contracts can optimize processes like compliances checks, payments, inventory control, reducing reliance on manual interventions [48].

2.3.1 Transparency, Traceability and Trust

Transparency, traceability, and trust are fundamental components of effective supply chain management, each one of these brings benefits to logistics.

Blockchain systems enhance transparency, which is crucial to bring trust among stakeholders, this transparency is achievable because each stakeholder, holds a shared ledger that stores all transaction in an untampered ledger. Since all stored transactions are immutable and timestamped, stakeholders gain a trustworthy audit trail of the product's history [49].

Furthermore, blockchain can enhance trust not only by making data transparent but by eliminating the need for a central authority, in supply chains companies often don't trust each others, so eliminating one more point of possible manipulation by the central authority enhances more trust between participants.

2.3.2 Smart Contracts and Automation in Logistics

Smart contracts play a pivot role in the implementation of blockchain systems in supply chain management, enabling automation of key process, by helping compliance, trigger payment events or manage inventory without manual oversight [49].

The most clear objective is in automating financial transactions and payments, normally when a shipment arrives and the paperwork is verified, a bunch of manual steps like invoicing, approval or bank payment, need to be done by an employee, this process can be slow and error-prone. Blockchain logs the receipt of goods, triggering a smart contract to release payment to the supplier or shipper, streamlining the process and ensuring timely payments, as noted in a 2024 industry report [48].

Another use of smart contracts is automating compliance and provenance verification, this is really useful in the pharma and food sector, where companies need to ensure that products are delivered meeting certain conditions, for example, certain meds need to be stored at a proper temperature.

Smart contracts can automatically check incoming data against compliance rules, an example of smart contract application comes from a pharmaceutical supply chain, the MediLedger blockchain network pilot, they used smart contract logic to verify drug identifiers and track ownership changes. This project demonstrates that blockchain systems can be used to validate the authenticity of product identifiers at each transaction and instantly confirm a drug's provenance back to the manufacturer [50]. Smart contracts enable automated verification, flagging counterfeit drug attempts and rapidly pinpointing affected lots for recalls. They act as automated compliance officers, enforcing traceability requirements in real time.

Similarly, smart contracts can help in inventory management by automating re-ordering or stock monitoring [48]. When a retailer's inventory goes below a defined threshold, a smart contract could automatically place a refill order with the supplier. Blockchain technology, when connected with other emerging technologies like IoT, has the potential to automate more manual tasks, imagine having an IoT sensor on a storage tank that reports the level of fuel inside, when the level goes below a specific value, smart contract can trigger scheduling a fuel delivery. Integrating IoT with blockchain, allowing sensor data, such as temperature, location, to feed into smart contracts. If a condition like a temperature excursion occurs, the contract can trigger alerts or insurance claims without human reporting, enabling responsive, event-driven automation that reduces supply chain delays [48].

Smart contracts enhance supply chain efficiency by codifying business rules into automated workflows on blockchain. They reduce the need for intermediaries, like banks, brokers or inspectors, they also reduce manual paperwork, thereby cutting costs and errors. However, implementing such automation across complex logistics networks remains challenging and typically requires all participants to agree on the smart contract terms and data standards.

2.3.3 Blockchain applied in logistics - Cases Studies

This section reviews real-world implementations from recent years in different industries, such us food/agriculture, global shipping and manufacturing. The cases drawn from academic analyses and

industry pilot reports, with the objective of showing how blockchain couples with smart contracts can be applied in practice, doing analyses of benefits and challenges each one of the project was faced.

Food and agriculture: IBM Food Trust and Walmart [51]

Walmart collaborates with logistic providers to enhance data capture for food distribution and tracking, making use of blockchain technology. Through devices that are connected to the blockchain, this system is capable of monitoring temperature and humidity during the distribution process. Allowing traceability at item level, improving food safety and response to contamination. The research explores blockchain's potential to enhance food safety, transparency, and trust in the supply chain, as seen in Walmart's pilot projects. The initiative promotes stakeholder collaboration to ensure product authenticity and quality throughout the supply chain.

Pharmaceutical Supply Chains: MediLedger and DSCSA Compliance [50]

In the pharmaceutical industry, drug traceability and anti-counterfeiting are critical concerns, and these have been explored by researchers for possible blockchain solutions. The most relevant project is MediLedger [50], a blockchain system for tracking prescription drugs. This project has been developed in partnership with the United States Drug Supply Chain Security Act (DSCSA). They developed a prototype blockchain application that logs each change of ownership of drug packages, the objective was gathering industry feedback on its performance and feasibility.

Mediledger pilot project shows that blockchain can meet the demanding requirements of pharmaceutical supply chains to satisfy DSCSA traceability rules. The pilot demonstrated satisfactory throughput, speed, and scalability, indicating its ability to accommodate the transaction volumes required for nationwide drug tracing while maintaining cost-effectiveness.

Global Shipping and Trade: Maersk-IBM TradeLens [52]

Global container shipping involves a complex logistic processes from carriers, ports, customs, logistic providers, all of these processes generate a massive paperwork and often suffer from a lack of visibility across the end-to-end journey. In 2018, Maersk and IBM collaborated to develop TradeLens, a blockchain platform designed to minimize human intervention and address the challenges associated with complex container shipping logistics. The platform's primary objectives were to achieve comprehensive traceability and digitalization on a global scale.

On TradeLens documents such as bills of landing, customs filings and cargo receipts are uploaded and timestamped on a blockchain, and with the usage of smart contracts the approvals and releases are automated, for example, when a container arrives in the port, the customs officer sign-off can be recorded on the blockchain, which automatically notifies the terminal and carrier that the cargo is clear to proceed, replacing this way phone calls or emails that can be easily forgotten causing delay of operations.

This platform, in mid-2019, had attracted five of the six largest ocean carriers, and with those and other shipping lines, the platform at that time had data for nearly two-thirds of global container freight volume.

Despite positive indicators, TradeLens ultimately faced challenges that led to its shutdown in 2023 [53]. Global industry collaboration was essential for the platform's success, but it remained elusive. Several major carriers and numerous freight forwarders exhibited reluctance to either join or integrate TradeLens extensively into their operational processes. According to Maersk, "the need for full global industry collaboration has not been achieved" [53], and therefore TradeLens did not reach the critical mass of usage to justify itself commercially.

Blockchain technology and smart contracts have show potential to transform supply chain management by enhancing transparency, efficiency and security [54]. There are several challenges to be addressed, but the benefits of this technology outweigh the costs. Since is a significant new technology with potential for evolution, it is likely to play an important role in shaping the future of supply chains.

2.4 Automatic Generation of Smart Contracts

As mentioned in previous sections, developing secure and correct smart contracts is a challenging and error-prone, that requires knowledge of both programming and blockchain specifics [45].

High-profile bugs, have caused million dollar losses, for example DAO hack referred by Das et al. [55] this attack caused losses rounding 60 million dollars. Moreover, domain experts (lawyers, business analysts, etc.) often struggle to translate legal or business requirements into code, leading to a gap between contract intent and implementation [56, 57].

The automatic smart contract generation aims to address developing efficiency, reducing human errors, and enabling the participation of non-programmers in the smart contract generation pro-

cess. In practice, smart contract automation can facilitate a broader blockchain adoption, thereby opening new oportunities in finance, supply chain management and the Internet of Things (IoT). This is achieved by enabling contracts to be generated from high-level descriptions while ensuring their correctness and security.

Developing smart contracts can be intricate and lengthy, necessitating profound knowledge of programming languages and blockchain infrastructures. To address this, Choudhury et al. (2018) [58] suggested a technique utilizing domain-specific ontologies and semantic regulations to automate the conversion of constraints from legal documents or protocols into smart contracts. This advanced approach employs parsing and abstract syntax tree handling methods to automatically produce smart contracts from defined requirements, easing the conversion of stipulations from legal contracts or standards, and ensuring repeatability. The method's success was showcased both by clinical study guidelines and vehicle leasing conditions. The suggested approach can potentially be adapted to different programming languages and blockchain systems, making it appealing for enterprises and people wanting to simplify the smart contracts development process. By automating this development, it becomes faster and more user-friendly, opening up avenues for various entities to leverage the advantages of smart contracts [58].

Aparicio et al. [59] introduce an approach that employs ontological models to support the effective deployment of smart contracts within an enterprise's blockchain framework. To illustrate the practical application of their method, the authors delve into a case study centered on the Rent-A-Car business model. Additionally, the research delves into the interplay between Blockchain-based smart contract and the DEMO Action Model, exploring the transformation of one into the other. Central to the paper's findings is the automated knowledge extraction from the DEMO Action Model, which aids generating Blockchain smart contracts.

Other previous work [60] discussed the usage of blockchain technology to streamline processes between mistrusting organizations. Their study underscores the use of smart contracts in DEMO Action Models and explores model-driven engineering for automatically generating these contracts. The paper promotes ontology-centric modeling techniques for precise smart contract specifications. While a connection between Solidity Concepts and DEMO Action Model concepts is presented, its optimization remains a topic for future research. The study uses a Rent-A-Car scenario for validation but suggests broader testing across different Action Models. In essence, ontology-driven

approaches can enhance data standards, business processes, and blockchain management, particularly in smart contracts, leading to better understanding and increased blockchain security.

Tong et al. (2022) [56] introduced an innovative approach to automate the creation of smart contracts. Their framework, termed AI-assisted smart contracts Generation (AIASCG), enhances collaboration and negotiation on contract clauses among contracting parties from varied backgrounds and languages. The method offers a universal portrayal of contracts using machine natural language (MNL), ensuring a mutual comprehension of contract commitments. Key to this proposal is an AI-driven technique for word segmentation named Separation Inference (SpIn). SpIn is pivotal to AIASCG, efficiently translating natural language sentences into intermediate MNL forms, minimizing the manual intervention in generating contracts.

In Tallyn et al.'s study [61], the application of smart contracts in four urban delivery models was examined, highlighting the balance between automation and the human touch necessary to maintain trust during deliveries. The research underscores the importance of interpersonal relationships in ensuring trust between delivery personnel and recipients.

2.5 Summary

We find a gap in the literature regarding practical solutions for the specific logistical needs of circular economy procedures using blockchain in collaborative micro-hubs. A scalable framework is needed to fully utilize this approach, enabling the creation and application of smart contracts for all stakeholders in a cooperative and user-friendly way. Even with the advancements reported in the domain of smart contracts, there remain significant barriers for non-technical individuals to easily enagate in the smart contracts creation process.

The vast majority of frameworks and structures employed are complex and often difficult to comprehend for the untrained eye. A relevant observation from the literature is that the methodology for elucidating and deploying these contracts largely relies on text-based explanations. This particular mode of presentation can prove to be particularly daunting for individuals lacking technical expertise, highlighting a significant challenge in current state-of-the-art.

Our research aims to reduce barriers surrounding smart contract generation through the evolution of visual component of smart contract design. By converting the traditionally text-heavy process into a visual specification, we anticipate that users can more intuitively participate on

the smart contract specification. A visually-driven methodology will reduce complexities and give confidence to the user. As they can see and interact with the visual elements, they become more engaged in both the creation and comprehension of the smart contract. Our emphasis on visual components seeks to not only simplify but also democratize the smart contract landscape, inviting participation from a broader spectrum of individuals, irrespective of their technical capability.

3 Context and Requirements

This chapter presents the MiCoLEC project and its relevance to the research. The chapter then introduces the DEMO methodology as the basis for modeling organizational rules, which will later be transformed into executable smart contracts (3.1). It introduces the platform's objectives (3.2) and describes the functional requirements (3.4) that guide its design. User profiles and user stories are detailed to illustrate the roles and interactions that the platform must support. The chapter concludes with the description of the system architecture (3.5), which structures the solution in components and services.

Implementing information systems aims to enhance organizational efficiency, but success depends on system functionalities and organizational attributes. DEMO uses the PSI theory of Enterprise Ontology to accurately represent the core of an organization in cohesive models and diagrams that mirror reality accurately [62]. We introduce new language elements in DEMO for smart contract creation from action models to improve readability and reduce errors.

In recent work/studies DEMO's Models were extended and validated with improvements on the representations of its Fact, Process and Action Models [63–66]. Regarding the Action Rule Specification (ARS) language, a comprehensive and explicit specification of logical and mathematical expressions, the specification of complex business rule logic and flow became feasible [62]. Such comprehensiveness is important in smart contracts, where minor mistakes can lead to substantial repercussions.

Our proposal appeals to a broader audience by incorporating DEMO Action Rules via a visual programming language, making it accessible to individuals with limited technical backgrounds. This approach improves the entire development process and innovates on the area of smart contract generation on blockchains by merging formal language with visual programming.

This dissertation report expands and details work presented in conference publications [3, 4], where we presented an initial prototype that gave us a valued initial insight on how to transpile DEMO ARS to smart contracts and also helped us with an initial understanding of what blocky visual components and grammar extension we should need to successfully automatically generate smart contracts within the logistics context, using the Hyperledger Fabric as our blockchain platform.

3.1 Design and Engineering Methodology for Organizations (DEMO)

We now will present the Action Model corresponding to recent evolutions of the grammar. For a more in-depth introduction to the DEMO methodology, the reader is referred to Appendix A.

3.1.1 DEMO Action Model

The AM of an SoI is the ontological model of its operation. For every internal actor role, it provides the rules that guide the role fillers in doing their work. The guidelines for responding to coordination events are called Action Rules (similar to business rules), the ones for performing production acts are called work instructions. An AM is represented by Action Rule Specifications and Work Instruction Specifications. The first ones guide actors in performing coordination acts, the second ones guide them in performing production acts. The AM is the solid foundation on which the other three models are standing. In a sense, they are already "contained" in the AM, they only need to be "extracted". Lastly, there is nothing "above" the CM.

DEMO Action Rules are the guidelines for managing events to which actors must react, or business rules. The Action Model of DEMO is not comprised by this set of rules alone, but also contains work instructions regarding the execution of production acts, both represented in the Action Rules Specification (ARS). Because work instructions are usually enterprise-specific, we will not elaborate on them. One should consider them as detailed instructions for accomplishing a certain production task, like the concluding of a rental contract. The Action ARS standard has evolved through time, starting with a pseudo-algorithmic language and culminating, in DEMO's specification language 4.5, in a definition which adheres to the EBNF (Appendix B), the international standard syntactic meta language, defined in ISO/IEC 14977.

These rules are sometimes referred to as business rules in modern usage. Every form of agendum that actors in a particular actor role must deal with when looping around their actor cycle has, in theory, a corresponding Action Rule. The "exception" states (declined, rejected, and the states in the revocation patterns) are mostly ignored because the response is much too dependent on the topical situation. One must at least produce the Action Rules that correspond with the coordination events in the basic transaction pattern, as seen on Figure 21.

The general form to represent an Action Rule is <event part> <assess part> <response part>. Figure 6 shows an example of this Action Rule specification form. What event (or collection of

concurrent events) is reacted to is specified by the event part. An Action Rule's assess portion is divided into three sections that correspond to the three validity claims: the claims to rightness, sincerity, and truth. The final section, the response, is broken down into an if-clause that outlines what must be done if the actor believes complying with the event is justifiable and, potentially, what must be done if it is not. This method of developing Action Rules enables the performer to stray from the "rule" if they believe it is acceptable, while also being held accountable for it [67].

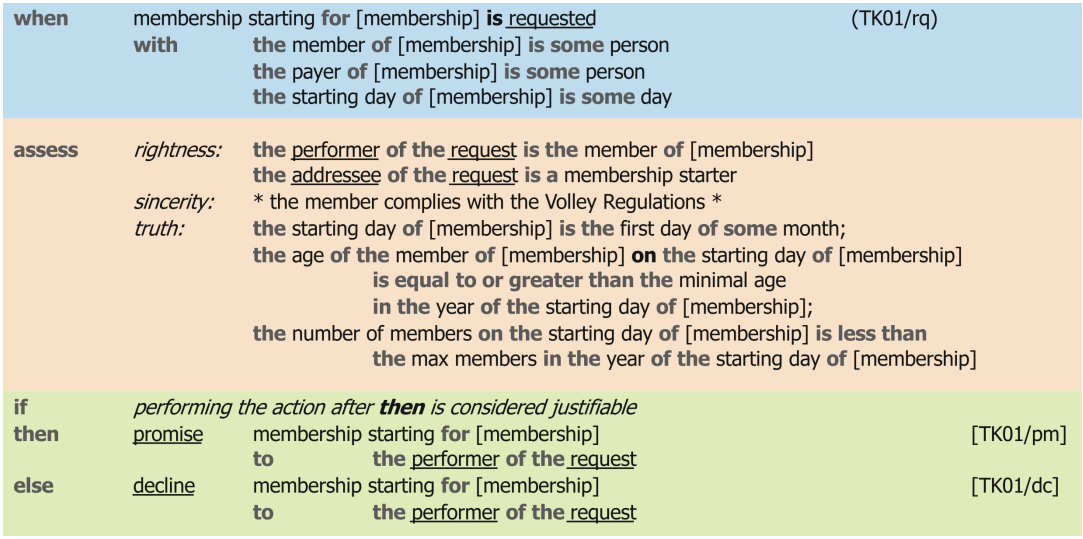


Figure. 6: Example of an Action Rule specification [68]

We consider this way of ARS to be ambiguous because, despite using a structured English syntax akin to that found in Semantics of Business Vocabulary and Rules, it does so in an imprecise manner that lacks some necessary ontological details to be used as the basis for the implementation of an information system. For instance, it lacks a method to deal with sets of actions or operators. Additionally, the current standard brings unneeded complexity since it includes a lot of extraneous details about three different forms of evaluation: fairness, sincerity, and truth [67].

3.2 MiCoLEC

The project "Micro-hubs Colaborativos para a Economia Circular" (MiCoLEC), aimed to establish a cooperative micro-hub of logistic where companies are expected to participate in a cooperative package delivery marketplace with the objective to improve their overall performance. This improvement is achieved by gains in efficiency levels (e.g. route sharing), less waste (e.g. deduplication of courier routes), and increase trust in cooperation through the platform's technology.

An innovative digital platform was proposed to ensure trustworthy and accountable information sharing among participants, in particular, this platform allows, for instance, package handoff between competing companies in a completely transparent and secure way, while ensuring that even if something goes wrong it is possible to trace the package movements and assess liabilities. To support this platform, MiCoLEC implemented a blockchain-based system that manages interactions between participants. This system enables the majority of the process to be executed on smart contracts, which transparently enforces fairness and that can not be tampered with.

Based on the problem MiCoLEC is solving, it was important to choose the right technology to support the platform. In order to do so, we identify the main problem characteristics and map them into technology requirements. In particular, given the nature of a platform with multiple participants lacking mutual trust, we focus on blockchain technology with its many variants and decided the best option for the MiCoLEC project.

After analyzing the functional requirements, the supporting technology had to meet at least the following requirements:

- Low or no fees on transactions. Having to pay a fee per transaction is a problem a delivery service may not support. These fees may be unpredictable and sometimes surpass the service value. The result would be an impairment in system scale. As soon as the system scales to high numbers of transaction throughput, the cost of those transactions would quickly become unbearable for any service provider.
- A scalable solution. A system may need to increase or decrease in performance and cost in response to changes in application and processing demands. It is important how easy these changes can be introduced in the system.
- Low barrier of entry for new users. No need to set up nodes or incur in high upfront costs. Roadblocks for new users (new logistics companies) to join the system will negatively impact the ability for the project to grow and expand.

3.3 Blockchain Solution analysis

The cost of a blockchain solution highly depends on various factors such as, features, complexity and type of blockchain. When using a public blockchain like Ethereum introduces transaction fees

known as gas, while private blockchain present cost in the form of infrastructure to operate the cluster.

On Ethereum, each transaction consumes a given amount of gas, a value can oscillate according to the transaction backlog, transaction data and required computing power. At the time of writing the fees for a simple ERC20 Token transfer is between 2 and 5 euros, under normal conditions, when the network is busy this value can increase to under 10 euros.

On private blockchains is required an infrastructure properly dimensioned to the expected performance. Implementing an infrastructure utilizing cloud services or hosted on-premises incurs monthly and upfront costs that fluctuate significantly based on the cluster capacity. Operating a blockchain cluster necessitates the presence of multiple nodes, RPC, block explorer, monitoring (data storage and visualization), and storage, which can collectively amount to between 1500 and 2000 euros for a modest setup.

The main advantage of private infrastructures on-premises regarding cost of ownership is predictability. While the on-demand nature of cloud services makes it more elastic, it also introduces the price variability. Nevertheless, a public blockchain that requires fees to process transactions can present high variability during periods of high network usage, moreover, as fees are incentives for processing a transaction, transactions with higher incentive are processed first.

The cost analysis was a sensible subject for the project since it can become very variable depending on the chosen path. Blockchain infrastructures are required to be highly distributed and scalable given the high computational requirements, moreover, for a project of this nature is expected a high number of transactions per operation or interaction, not only that, but also, high throughput. Another requirement to take into account is storage, since it's something that will increase over time and thus, hard to be dimensioned without initial usage samples, for that reason, a resilient distributed file system able to grow overtime was taken in consideration.

Due to the requirements for the project and the previous cost overview, it becomes clear that paying fees for each transaction or operation was not feasible in the long term due to fee volatility and transaction requirements. Also, requiring every participant to pay fees for each interaction may be an adoption deterrent.

The cloud or on-demand route offers the least resistance to entry, along with more predictable costs and without the upfront cost of an on-premises infrastructure.

Having a permissioned/private blockchain removes the need for fees and opens the path for more creative practices regarding token incentives and penalties. Using cloud or on-premises are the most attractive solutions regarding cost predictability, although there certainly is some up-front cost, it lowers the barrier to entry and the infrastructure investment is gradually reduced over time, with the benefit of not being affected by a volatile fee structure. Also, there is more control and ownership over the infrastructure with much lower vendor lock-in.

Considering the study carried out on blockchain technology and the MiCoLEC context, we now present an analysis of possible solutions for the project.

3.3.1 Blockchain Systems Analisys

From the different systems available, we identified a group of solutions that could be used as the blockchain component for MiCoLEC. In this subsection, we briefly describe each one of them. The information listed here is mostly retrieved from each technology's official documentation.

- **Hyperledger** [69]: An open source project created to support the development of blockchain-based distributed ledgers. Hyperledger consists of a collaborative effort to create the needed frameworks, standards, tools and libraries to build blockchains and related applications. Hyperledger Fabric was founded by the Linux Foundation, which is Hyperledger's framework with the most use cases and support. This component implements complex permissioning by, besides having validators, allowing each user to have a defined role, restricting the actions they can perform on the blockchain. All participants have known identity which is validated against the organization's identity management system. There are no anonymous or pseudonymous users. There is no PoW algorithm and crypto mining in Fabric, which allows for high scalability and fast transactions. One of the most interesting characteristics of Hyperledger is its modular architecture that allows for the development of custom plug-in components. Hyperledger is supported by one of the richest development communities in the space.

- **Quorum** [70]: an open source blockchain protocol specially designed for use in a private blockchain network, where there is only a single member owning all the nodes, or, a consortium blockchain network, where multiple members each own a portion of the network. Quorum Smart contracts are written in the Solidity language and the Raft protocol is used as the consensus mechanism in Quorum. This choice is targeted at higher transaction throughput rates when compared with Proof-of-work approaches. However, Quorum's channel-based approach

to privacy presents challenges for privacy and scalability as use cases become more complex. In addition, Quorum does not require a built-in cryptocurrency because consensus is not reached via mining. As a consequence, it is not possible to develop a native currency or a digital token with Quorum, which for our own scenario is a dealbreaker.

– **Multichain** [71]: was developed by Coin Sciences and is a fork of the Bitcoin blockchain. However, unlike Bitcoin, MultiChain allows users to configure several parameters such as the permissions to access the network, the privacy of the chain, the maximum block size, and the mining incentive. MultiChain supports a variety of programming languages such as Python, C#, PHP, Ruby or JavaScript. It focuses on the two strong use cases: the asset ownership lifecycle (issuance, payment, exchange, escrow, retirement) and General immutable data storage. It is very easy to install, configure and create a network MultiChain since you don't need to write any code and can get started immediately through easy-to-use APIs. Multichain simplicity may attract developers and organizations but the inability to write custom smart contracts prevents us from considering it as an option.

– **Corda** [72]: is written in the Kotlin programming language and supports development both in Kotlin and Java. Corda exists in two main editions. There is an open-source edition that is free for personal and commercial use called Corda, and the enhanced paid edition called Corda Enterprise. Corda Enterprise offers additional performance enhancements, such as higher computational capacity for large-volume transactions. While Corda is used in a variety of industries, the majority of its customers come from the finance, banking, insurance, and capital markets sectors. Corda's private blockchain features are particularly relevant for the companies in these sectors, as data confidentiality is highly important for their operations. The biggest advantage of Corda for businesses is the ability to protect the privacy of transactions. Corda's most common use cases include inter-organizational cooperation. By creating a blockchain-based network on Corda, businesses can significantly improve cooperation efficiency and cut down on the cost of interacting. For example, a Corda network of insurance companies, brokers, and re-insurers can streamline claims processing, data verification, mutual payments, and other business processes. Corda brings the benefits of blockchain to finance-related industries while ensuring that confidentiality and privacy, so much heralded by companies in these industries, are not compromised. Considering our use case is in the logistics domain, it is hard to assess the feasibility of using Corda since there is not much previous experience and use cases to

draw from. Moreover, the opaque pricing from R3 (the company behind Corda) makes it really hard to take into consideration at such an early design stage due to the increased difficulty in predicting what the system may demand from the blockchain component.

- **Ethereum** [73]: was created to address some of the shortfalls of Bitcoin. While Bitcoin is great for storing wealth (BTC is the most secure cryptocurrency in the world) it lacks complex functionality. You can send and receive transactions and execute some other essential functions, but smart contracts are not supported. That's where Ethereum comes in. Ethereum offers a high level of customization so that developers can create custom products. Ethereum has been developed as a permissionless, public blockchain, in which every smart contract can be programmed in connection with decentralized applications (dApps). For this, a virtual machine (VM) is provided on the blockchain, for which a fee must be paid depending on the effort required to execute the programming code. The most used programming language for Ethereum is Solidity. Ethereum is the second most decentralized cryptocurrency in the world, after Bitcoin. It has the largest developer community in the world, even larger than Bitcoin's. This gives Ethereum a tremendous advantage over other protocols. When building an app on Ethereum, you can instantly connect it to hundreds of other protocols that already exist. In the Ethereum community, this is known as money legos. As great as Ethereum is, the platform certainly is not perfect. As we can see with Bitcoin and Ethereum, decentralized protocols tend to be slow. Bitcoin has average speeds of 7 TPS (Transactions Per Second), while Ethereum has a speed of 15 TPS. That's double Bitcoin's speed, but it's not nearly enough. The Ethereum coin that powers the network is officially named Ether and popularly known by its ticker symbol — ETH. The gas fee on the Ethereum network remains one of the biggest challenges, and it is also affecting the Ethereum blockchain scaling. In fact, when the gas fee hits high, the Ethereum network has been redirecting users to other platforms, which shows how serious of a problem those fees can be.
- **Hedera** [74]: This distributed ledger technology that uses a specific transaction handling and voting protocol to make it faster and more energy-efficient when compared with Bitcoin or Ethereum. That approach is called Hashgraph and instead of grouping data into blocks, a consensus protocol works for each transaction determining if such particular transaction is added to the ledger or not. This approach speeds up transaction times, making a Hashgraph network capable of handling up to 250,000 transactions per second. This speed is currently

throttled to 10,000 TPS on the Hedera Hashgraph, but it can get lifted if the need arises. Another benefit of its consensus protocol is that transactions get confirmed in about 3 to 5 seconds. This puts Hedera way beyond the 10 to 60-minute blockchain confirmation time-frames and sets it on par with credit card companies. Hashgraph uses what is called asynchronous Byzantine fault tolerance to maintain a secure network. Byzantine fault tolerance takes the potential unreliability of the network's nodes into consideration when reaching a consensus, to avoid a damaging system collapse. The system is also protected against DDoS and Sybil attacks. Hashgraph uses just about 0.0002 KWh per transaction, making it extremely more energy-efficient and environmentally friendly than most blockchains. Hedera Hashgraph's transaction fees are also very low and start from $0.0001, depending on exactly what you need to get done on the system. The costs are significantly lower than the $15+ that popular blockchains charge per transaction.

Based on the analysis done, we decided that the two most suitable systems to be used for MiCoLEC blockchain component are Hyperledger and Hedera. They both support customized smart contracts and a controlled environment. This was crucial to lower the entry barrier for newcomers to MiCoLEC platform. In this way, only a subset of participants will be required to provide infrastructure to support the platform, and newcomers may try the system without incurring in high upfront costs. Later on, each participant will be able to increase their stake in the system by providing infrastructure and contributing to the decentralization and robustness of the solution. The rejection of public blockchain technology is grounded on the assumption that a system such as MiCoLEC will scale in the number of transactions (packages) running in the system. As a consequence, any solution that has costs per transaction was not viable in this scenario.

Another important fact was that both Hyperledger and Hedera are supported by strong companies and consortiums, which typically assures continuous development and bug fixing as well as good developer support. These aspects were critical for a project such as MiCoLEC, which was expected to run for two years and still had a number of open challenges to address. Having a customizable, extendable, and mature system as the foundation for such endeavor was essential to avoid discovering technological limitations too late in the development efforts.

3.3.2 Proposed solutions and brief experimental evaluation

From the analysis and research up to now, we concluded that the two most suitable systems for the MiCoLEC blockchain component are Hyperledger and Hedera. Both support customized smart-contracts and a controlled environment. This was a very important to lower the entry barrier for newcomers to the MiCoLEC platform.

Since the goal was to quickly assess potential major differences in maturity of both systems, we devised a simple test where we wanted to create a custom token from scratch and get a sense of the difficulty of such a task. We have locally deployed both systems and used their testnet setup for the experiments. Both systems were fairly easy to set up and both had good documentation. We then proceeded to create a custom token on both platforms. The main difference between Hyperledger and Hedera were the SDKs provided by the latter, which provided a slightly smoother experience. However, Hyperledger seems to better support customization, which for the MiCoLEC project was paramount. Additionally, both these technologies are supported by an extensive community of developers and are accompanied by sound documentation.

Although these were very limited tests, they gave us confidence that choosing any of these platforms will allow us to design and develop MiCoLEC without major roadblocks. The edge pends towards Hyperledger due to the broad scope of use cases already running there and for the highly customized design.

Following the different data points and analysis presented, our decision was to move forward with Hyperledger as the blockchain/distributed ledger component for MiCoLEC. The goal was to provide an extendable and customizable platform that allow the implementation of MiCoLEC, but also leave room for improvements and platform evolution in the future. Additionally, the maturity of Hyperledger, which is in fact one of the most mature technologies in the space, provides the necessary support for innovation.

3.4 Objectives and Functional Requirements

In this section we present the main objectives and the functional requirements of MiCoLEC project, which are specified through user stories, and provide comprehensive descriptions of the expectations of each user profile from the platform.

3.4.1 Objectives

The MiCoLEC platform was designed to support collaboration among multiple actors in the logistics chain, with a particular focus on promoting circular economy practices. Its objectives are centered on the creation of a digital marketplace where logistics operators can offer and bid for parcel deliveries according to their operational interests, receiving or paying rewards depending on the type of transaction in which they participate. The platform also extends these opportunities to couriers, who may bid for deliveries and obtain rewards upon successful completion of their work. End customers are given the possibility to engage directly in reverse logistics by requesting the collection of goods intended for reuse or recycling whenever they receive a parcel. In addition, circular economy operators are supported in offering rewards for parcels that contain goods of interest, thus stimulating reuse and recycling activities. Collectively, these objectives contribute to establishing a collaborative and transparent environment that reduces operational costs, facilitates wider geographic coverage, and promotes sustainable practices in urban logistics.

3.4.2 User Profiles

The MiColEC platform serves five user profiles: platform administrators, logistics operators, couriers, circular economy operators, and end customers.

1. **Platform Administrators** have superuser privileges in the platform, which allow them to manage all operational data, except for information concerning the free trading of parcel deliveries.
2. **Logistic Operators** are certified express delivery companies that can offer and bid parcel deliveries.
3. **Couriers** are individuals or companies that are not certified to make express deliveries but are qualified to deliver parcels. These users can only bid parcel deliveries.
4. **Circular Economy Operators** are companies certified to recycle or reuse goods.
5. **End customers** are individuals or companies that are recipients of parcels.

3.4.3 User Stories

A user story in requirement engineering is a user-centered artifact that captures a functional requirement from the perspective of the user. We next preset all specified user stories for each profile described above.

Platform Administrators (PA)

a) As a PA, I want to be able to login to the platform.

b) As a logged in PA, I should be able to access a management dashboard and have access to a list of participants in the platform and parcels status.

c) As a logged in PA, I want to be able to add a new individual or organization to the platform and define access credentials so they can use the platform.

d) As a logged in PA, I want to be able to revoke platform access to an organization and correspondent credentials.

e) As a logged in PA, I want to be able to define a formula to calculate a recommended price for a delivery.

f) As a logged in PA, I want to be able to check the status of bitcircles in the platform (e.g. total amount)

g) As a logged in PA, I want to be able to create and give bitcircles to platform users.

h) As a logged in PA, I want to be able to check the status of the platform (parcels on auction, parcels transactioned, completed deliveries and pending deliveries).

i) As a logged in PA, I want to be able to define a bitcircle reward for parcel deliveries and circular economy parcel deliveries.

Logistic Operator (LOps)

a) As a LOp, I want to be able to register (or request access) to the platform to receive access credentials.

b) As a logged in LOp, I should be able to access a management dashboard and have access to the most recent parcel offers and bids as well as to some important statistical data of my activity in the platform.

c) As a logged in LOp, I want to be able to get an API key that allows my organization's systems to make requests to the platform's API.

d) As an accredited LOp, I want to be able to download the platform's backend code to be able to deploy and run it on my infrastructure.

e) As a logged in LOp in the backend, I want to check the platform's recommended delivery price for a parcel or group of parcels.

f) As a logged in LOp in the backend, I want to be able to add a request for delivery of a parcel or group of parcels to the platform defining metadata about the delivery.

g) As a logged in LOp in the backend, I want to be able to accept a bid made to a parcel delivery offer I have made.

h) As a logged in LOp in the backend, I want to be able to accept a request of a circular economy parcel delivery made by an end user that was previously accepted by a circular economy operators.

i) As a logged in LOp in the backend, I want to be able to list available requests for delivery and bid on them at a certain price/bitcircle pair of values.

j) As a logged in LOp in the backend, I want to be able to assign a delivery to a user in my organization.

k) As a logged in LOp in the backend, I want to be able to view all parcel offers and bids I have made.

l) As a logged in LOp, I want to list my pending deliveries and list each delivery details (recipient, address, etc).

m) As a logged in LOp, I should be able to check my bitcircles balance.

Courier

a) As a Courier, I want to be able to register on the platform in order to offer my services.

b) As a logged in Courier, I should be able to access a management dashboard with most relevant information to me, e.g. latest bids status, latest undelivered parcels, and awards received.

c) As a logged in Courier, I want to list my pending deliveries and list each delivery details (recipient, address, etc).

d) As a logged in Courier, I should be able to check my bitcircles balance.

e) As a logged in Courier, I want to list available open deliveries and bid with a price/bitcircles pair.

f) As a logged in Courier, I want to validate that the declared content of a CE parcel delivery corresponds to the real content of the parcel.

g) As a logged in Courier, I want to be able to accept a request of a CE parcel delivery made by an end user that was previously accepted by a circular economy operators.

h) As a logged in Courier, I want to be able to view all parcel offers and bids I've made.

End User

a) As an end user, I want to be able to register on the platform to be able to access its services.

b) As a logged in end user, I should be able to list my pending deliveries and correspondent estimation of delivery.

c) As a logged in end user, I should be able to check my bitcircles balance.

d) As a logged in end user, I should be able to ask for the delivery of a CE parcel delivery for an existing delivery and use bitcircles for that operation.

e) As a logged in end user, I want to be able to confirm the reception of a delivery.

f) As a logged in end user, I want to be able to make an offer for a parcel delivery or a CE parcel delivery.

Circular economy operators (CEOp)

a) As a CEOp, I want to register on the platform.

b) As a logged in CEOp, I should be able to access a management dashboard with most relevant information to me, e.g. parcels I am about to receive, received parcels, awards I gave for receiving parcels.

c) As logged CEOp, I want to list what kind of objects I am willing to receive and the corresponding award I am available to give for them.

The set of user stories presented specifies the functional behavior expected from the MiCoLEC platform, clarifying the roles of administrators, logistics operators, couriers, circular economy operators, and end customers in the collaborative environment. These stories capture the interactions that must be supported by the system. The next section introduces the overall architecture of

MiCoLEC platform, and a detailed architecture of the solution that integrates blockchain, smart contracts, and supporting services, which was the focus of this dissertation's project.

It is worth noting that most of the functionalities regarding user stories above was developed by another project colleague, while only the developments requiring the use of the blockchain component were under my responsibility. The smart contract component in MiCoLEC project has the following main functional requirements:

- The system shall allow users to create and participate in auctions of parcels.
- The system shall close automatically the auctions that are expired.
- The system shall allow users to query auctions: all, by id, by parcel or state.
- The system shall allow user to place bids on open auctions.
- The system shall allow user to query bids: all, by auction or by participant.
- The system shall allow users to create new parcels.
- The system shall allow users to query parcels: all, by id or by state
- The system shall allow users to transfer bitcircles between accounts.
- The system shall create automatically create wallets every time a user is created on the platform.

The list of these developments can be found in Table 3.

3.5 Architecture

The initial architecture proposal, as depicted in Figure 7, is a multi-layered approach. At the center of the proposed design is the blockchain component. The blockchain serves as the data repository log where all transactions, that is deliveries, order status, etc. are recorded. This component, which will be described in more detail in the following section, provides much-needed transparency, privacy and resiliency to all platform participants.

The deployment of the blockchain component in permissioned private mode, where the infrastructure is owned and deployed by the project consortium. In this way, ownership and transaction fees become much more manageable in the long run. The downside is the greater cost of ownership at the beginning and possibly greater barriers to entry for new participants. New participants can

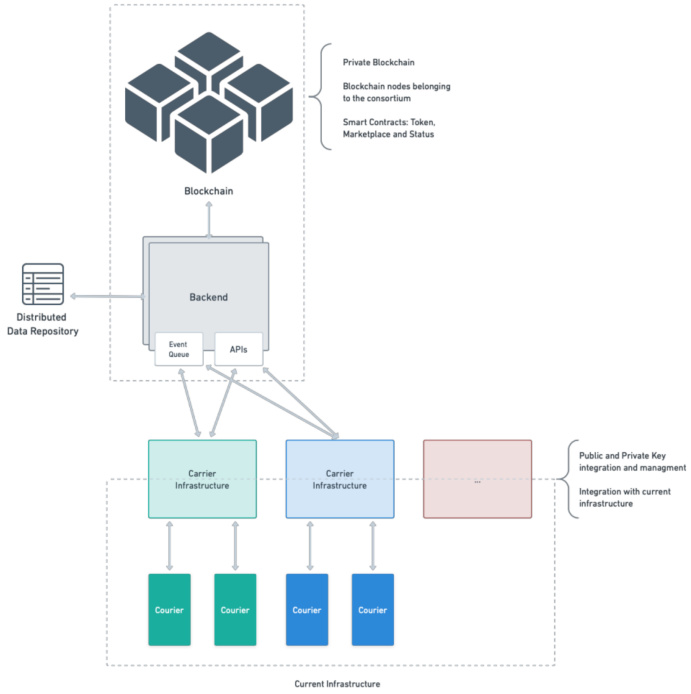


Figure. 7: MiCoLEC Architecture Diagram

either join the consortium, meaning they must run their blockchain node helping support, maintain and manage the platform, or, on the other hand, join the platform as a simple participant and to do so, connect to the deployed infrastructure and consume the exported APIs and integrate them with their own existing infrastructure. Coupled with the blockchain node there is a backend instance managed by each participant of the consortium, and deployed in their own infrastructure. The role of the backend is to abstract the intricacies of the blockchain and provide an easier to integrate and a more familiar interface, further lowering the barriers to entry for new participants. The backend is responsible for interfacing with the blockchain, for interfacing with a Distributed Data Repository where auxiliary and intermediate data can be stored. In addition, the backend export interface APIs and Event Queues that will be consumed by the participants of the platform.

As mentioned previously, new carrier participants just need to consume and integrate the exported APIs into their own existing infrastructure. In this way, there is minimal intrusion and changes to the way they currently operate, even integrations with Web/Mobile apps should be seamless with the proposed design.

To better visualize the architecture, we will now analyse the in detail the MiCoLEC project architecture, represented on Figure 8.

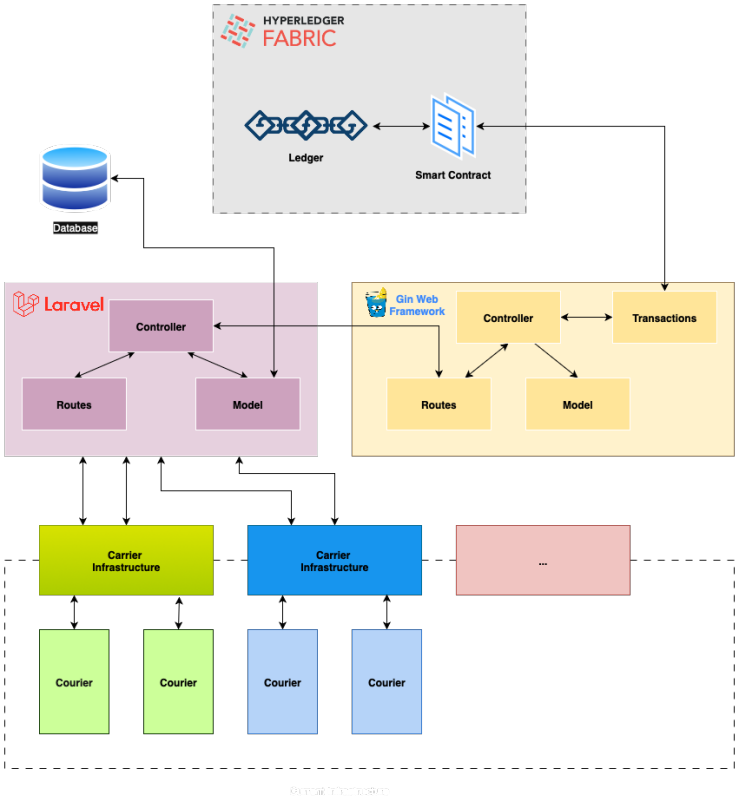


Figure. 8: MiCoLEC Detailed Architecture

The implementation is composed by a Laravel API, Gin Web Framework and a Hyperledger Fabric blockchain system.

Laravel is a php framework that provides a structured for creating web applications. Laravel offers a good developer experience with features like thorough dependency injection and an expressive database abstraction layer. On MiCoLEC Laravel Framework was used as an API to provide endpoints that allow participants to interact, either using a provided React front-end application from MiCoLEC project or their own front-end application. The Laravel API establishes a connection to an external database where data that is not pertinent to be stored on the blockchain is stored. Some information is also duplicated in both database systems, such as parcel information.

The Gin API (Go based API) is responsible for invoking the methods defined in the deployed smart contracts on the Hyperledger Fabric network. The creation of this API was necessary because at the time of implementation the only official SDK that allows to make request through http request is "fabrick-sdk-go", this SDK was released on GitHub by the Hyperledger developers [75].

The last component is a Hyperledger Fabric blockchain system, where the smart contracts are deployed, as well as the imutable and transparent states of the system. Hyperledger Fabric's smart contracts can be programmed using Chaincode. Chaincode can be implemented in various programming languages, including Go, Java, and JavaScript. Go is the primary language for Chaincode smart contracts, and its documentation is more comprehensive. Therefore, we have chosen to program the smart contract using Go. It is also in this blockchain component that the ledger is stored, representing the place where all transactional data resides and is persistently maintained.

4 Implementation

This chapter describes the developed solution. It begins with the specification of the APIs (4.1, 4.2) required to interact with the blockchain system. It then details the design and implementation of smart contracts (4.3) in Hyperledger Fabric, focusing on the generation of Chaincode from DEMO Action Rules (4.4). The proposed extention to EBNF grammar, transpilation method, and file structure are presented, along with examples that demonstrate the transformation process.

4.1 Laravel API

On MiCoLEC Laravel Framework was used as an API to provide endpoints that allow participants to interact, either using a provided React front-end application from MiCoLEC project or their own front-end application.

For this thesis, the relevant part are the requests that are directed to the blockchain, so we will stay on this scope. When requests are done for the blockchain, the request must follow a dedicated URL, in Laravel this behavior was implemented by the implementation of a route, as show next:

Code 1: Blockchain route

```
Route::match(['get', 'post'], '/blockchain/{endpoint}', [BlockchainController::class,
    'CallAPIHyperledger'])->where('endpoint', '.*');
```

This route delegates the request to the BlockchainController, which is responsible for handling the interaction with the blockchain. The controller processes both GET and POST requests, sending the participant's identifier to ensure that operations are performed within the correct user context.

If the request is a GET, the participant's ID is appended to the query parameters before forwarding the request. For POST requests, the payload is merged with the user identifier and sent to the Gin API. Invalid methods return an error response. The controller also handles specific cases, such as excluding certain endpoints (e.g., parcel operations), and performs additional logic after successful blockchain transactions. For instance, in the case of auctions, the controller updates parcel records in the local database to reflect the state change once the blockchain confirms the transaction.

Code 2: Blockchain Controller

```
class BlockchainController extends Controller
```

```
{
    public function CallAPIHyperledger(Request $request, $endpoint)
    {
        $logged_participant = Participant::where('user_id', $request->user()->id)->first();
        $blockchainUrl = env('MICOLEC_API_BLOCKCHAIN_URL');
        $url = $blockchainUrl . '/' . $endpoint;
        $queryParams = $request->query();
        $httpMethod = $request->method();
        if (str_starts_with($endpoint, "parcel")) {
            abort(404);
        }
        if ($httpMethod === 'GET') {
            // Add participant ID to query parameters
            $queryParams['user_id'] = $logged_participant->id;
            $response = Http::get($url, $queryParams);
        } elseif ($httpMethod === 'POST') {
            $response = Http::post($url, array_merge($request->all(), ['userId' =>
                $logged_participant->id]));
        } else {
            return response()->json(['error' => 'Invalid HTTP method'], 400);
        }
        $responseData = $response->json();
        if (!isset($responseData['data']) || !isset($responseData['data']['data'])) {
            // Handle the case where 'data' or 'data'['data'] does not exist in the
                response
            return response()->json($responseData, $response->status());
        }
        if ($endpoint == 'auction' && $response->status() == 201){
            // CALL UPDATE PARCEL
            $parcels = $responseData['data']['data']['parcels'];
            foreach ($parcels as $parcel) {
                try {
                    $parcel_record = Parcel::find($parcel);
                    if ($parcel_record === null) {
                        return response()->json(['message' => 'There is no Parcel with that
                            id.'], 404);
                    }
                    $parcel_record->state = 'Auction';
                    $parcel_record->update();
                }catch (\Throwable $th) {
                    echo $th;
                }
            }
        }
        return response()->json($responseData, $response->status());
    }
}
```

The implementation this route and controller in Laravel API, opens a connection between the MiCoLEC front-end and the Gin API, in the next step we will look into the Gin API and why this layer it's important.

4.2 Gin API

The Go-based API, whose source code is provided in a online repository [76], is responsible for invoking the methods defined in the deployed smart contracts on the Hyperledger Fabric network, the creation of this API, as mention before, was necessary because the only official SDK that allows to make request through http request is "fabrick-sdk-go" [75].

This layer acts as the bridge between the backend (Laravel API) logic and the Hyperledger Fabric Smart Contract, it also ensures that every relevant business operation in the application corresponds to a validated and immutable transaction on the distributed ledger.

This solution provides a mechanism to route Laravel requests to the blockchain layer while ensuring data consistency and correctness across the system, that is important because once data is added to the blockchain is permanent and cannot be undone.

On the next Table 3, we will present all the routes develop on this API, with a quick description of each one of them, after that we will look into an example.

Table 3: Gin API implemented Routes

Http Method	**URL**	**Controller Method Name**	**Description**
POST	$\backslash auction$	CreateAuction	This request is used to create new auctions, for that purpose its expected to receive the auction data, parcels that will be added to the auction and user that is creating the auction.
GET	$\backslash auction$	GetAuctions	This route was develop to query auctions by state, parcel, or a full view of available auctions on the blockchain. Pagination was implemented on the API due to time constraints, preventing its implementation on the smart contract.

GET	*auction**:id*	GetAuctionByID	In order to retrieve an auction information, this route was implemented, it receives only one query parameter that is the auction id.
POST	*bid*	CreateBid	This route is responsible to place bids on auctions, it receive the bid properties, including the user id, and do the request to the smart contract method.
GET	*bid*	GetBidForAuction	This request is use to get the bids done in specific auction, the request needs to receive as query parameter the auction id. This route also have pagination that can receive the pagination items per page and page number.
GET	*bids*\mybids	GetParticipantBids	This route was developed to retrieve all bids done by the user, the user id is send using query parameter. In this route pagination was also implemented.
GET	*bids*	GetAllBids	This route returns all bids done in the platform.
POST	\parcel	CreateParcel	This route is used to create parcels, this route receive the parcel and user and then make the request for the specific smart contract method.
GET	\parcel	GetParcels	To retrieve parcels this request have to ways, if it do no receive a query parameter of state, it will retrieve all parcels on the systems. Otherwise if it have a state value on query parameter, it will only retrieve parcel on a given parcel state.
POST	\wallet	CreateWallet	This method is used to generate a wallet for a participant.
GET	\wallet\:id	GetWalletByParticipantID	This request retrieves the user wallet, it receives as query parameter the user id.
GET	\wallet\transactions\:id	GetParticipantBitcircle Movements	This request is used to query user movements, this request also uses pagination.

POST	\wallet\transaction	TransferBitcircles	This method is the one used to transfer bitcircles between participants. This method expect to receive the following information: sender id, receiver id, bitcircle amount, a boolean that represent if this transfer is a reward or not, and finally a description of the transfer.

Taking in consideration that the flow for different request is similar we will present the flow of creation of an auction, and the rest can be consulted on the Gin API project that is shared in the online repository [76].

The first thing we do was create a model for an auction, in our case an auction should have the following properties: ID, start date, end date, maximum accepted licitation, state and participant ID. The following Go code is the model implementation of the auction model.

Code 3: Gin Framework Auction model

```
type AuctionState string
const (
   AuctionClosedNoBids Status = "CLOSED NO BIDS"
   AuctionClosedBids Status = "CLOSED BIDS"
   AuctionOpen       Status = "OPEN"
)

type Auction struct {
   ID                       string      `json:"id"`
   StartDate                time.Time   `json:"start_date"`
   EndDate                  time.Time   `json:"end_date"`
   MaximumAcceptedLicitation float32  `json:"maximum_accepted_licitation,omitempty"`
   State                    AuctionState `json:"state"`
   ParticipantId            int         `json:"participant_id"`
}
```

To receive requests we create routes, in the case of auction creation is a POST request, using the URL "/auction", the implementation of this route is presented below:

Code 4: Gin Auction Creation route

```
router.POST("/auction", controllers.CreateAuction())
```

The routes delegate the request to the controller in order to make the request to the blockchain, for that purpose we create a controller method that is responsible for the auction creation.

Code 5: Gin Framework Auction Creation Controller

```
func CreateAuction() gin.HandlerFunc {
   return func(c *gin.Context) {
      var input struct {
         RequestAuction        models.Auction `json:"auction"`
         RequestParcelHasAuction []int       `json:"parcel_has_auction"`
         RequestUserId         int           `json:"userId"`
      }

      if err := c.ShouldBindJSON(&input); err != nil {
         c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
         return
      }


      if validationErr := validate.Struct(&input.RequestAuction); validationErr != nil {
         c.JSON(http.StatusBadRequest, responses.Response{Status: http.StatusBadRequest,
             Message: "error", Data: map[string]interface{}{"data":
             validationErr.Error()}})
         return
      }

      dateNow := time.Now()
      formattedDate := dateNow.Format("2006-01-02T15:04:05Z")
      layout := "2006-01-02T15:04:05Z"
      parsedTime, err := time.Parse(layout, formattedDate)
      if err != nil {
         c.JSON(http.StatusBadRequest, responses.Response{Status: http.StatusBadRequest,
             Message: "error", Data: map[string]interface{}{"data": err.Error()}})
         return
      }

      newAuction := models.Auction{
         ID:                      uuid.New().String(),
         StartDate:               parsedTime,
         EndDate:                 input.RequestAuction.EndDate,
         MaximumAcceptedLicitation: input.RequestAuction.MaximumAcceptedLicitation,
         State:                   input.RequestAuction.State,
         ParticipantId:           input.RequestUserId,
      }

      auctionParcels := []models.AuctionHasParcel{}

      for _, parcel := range input.RequestParcelHasAuction {
         currentParcel := models.AuctionHasParcel{AuctionID: newAuction.ID, ParcelID:
             parcel}
         auctionParcels = append(auctionParcels, currentParcel)
      }
```

```
        err = validateAuction(newAuction)
        if err != nil {
            c.JSON(http.StatusBadRequest, responses.Response{Status: http.StatusBadRequest,
                Message: "error", Data: map[string]interface{}{"message": err.Error()}})
            return
        }

        result, err := transactions.AddAuctionTransaction(newAuction, auctionParcels)
        if err != nil {
            fmt.Println("Failed to submit transaction: ", err.Error())
            c.JSON(http.StatusBadRequest, responses.Response{Status: http.StatusBadRequest,
                Message: "error", Data: map[string]interface{}{"message": err.Error()}})
            return
        }

        data := base64.StdEncoding.EncodeToString(result)
        decodedData, err := base64.StdEncoding.DecodeString(data)
        if err != nil {
            c.JSON(http.StatusBadRequest, responses.Response{Status: http.StatusBadRequest,
                Message: "error", Data: map[string]interface{}{"message": err}})
            return
        }

        var res ErrorResponse
        err = json.Unmarshal(decodedData, &res)
        if res.ErrorCode != 0 {
            c.JSON(res.ErrorCode, responses.Response{Status: http.StatusBadRequest, Message:
                "error", Data: map[string]interface{}{"message": res.ErrorMessage}})
            return
        }

        var resultAuction struct {
            Auction models.Auction `json:"auction"`
            Parcels []int          `json:"parcels"`
        }

        err = json.Unmarshal(decodedData, &resultAuction)
        if err != nil {
            log.Println("ERROR: ", err)
            c.JSON(http.StatusBadRequest, responses.Response{Status: http.StatusBadRequest,
                Message: "error", Data: map[string]interface{}{"message": err}})
            return
        }

        c.JSON(http.StatusCreated, responses.Response{Status: http.StatusCreated, Message:
            "success", Data: map[string]interface{}{"data": resultAuction}})
    }
}
```

The first step of this controller is to bind the JSON object into the corresponding models, necessaries for the creation of an auction: auction properties, parcels attached to auction and user id that is creating the auction. In case of fail doing this binding, an error is thrown with the corresponding error message.

After the binding and validations, we create a timestamp information on server, this timestamp information has to be created on server because, since we are dealing with a distributed system if we try to add the timestamps on smart contract, different nodes will have different timestamps which leads to errors, because all nodes probably will have different timestamps for the same auction.

Next is created the relations between auction and respective parcels. After that the process is straight forward, we check business rules and send the request into the transaction component that is responsible to make the blockchain request. After receive the response from the blockchain the controller respond to the initial request that is done by the Laravel API.

The transaction component as said previously is responsible to make the connection to the smart contract deployed on Hyperledger Fabric. The code is simple on this component because all the hard work is done on the controller component.

Code 6: Gin Auction Transactions Component

```
func AddAuctionTransaction(auction models.Auction, parcels []models.AuctionHasParcel)
    ([]byte, error) {
   log.Println("--> Evaluate Transaction: Add Auction")
   contract := GetContract()

   args := []string{}
   parcelsBytes, err := json.Marshal(parcels)
   if err != nil {
      return nil, err
   }
   args = append(args, string(parcelsBytes))

   auctionBytes, err := json.Marshal(auction)
   if err != nil {
      return nil, err
   }
   args = append(args, string(auctionBytes))

   result, err := contract.SubmitTransaction("ParcelDeliveryAuctionStart", args...)
   return result, err
}
```

In the code above it is possible to check the following steps done by this component:

1. Get the contract object, this makes the connection to the blockchain smart contract.
2. Convert the list of parcels into a JSON.
3. Add parcels JSON string to the arguments list.
4. Convert auction object to JSON.
5. Add auction JSON string to the arguments list.
6. Call the smart contract function ParcelDeliveryAuctionStart, passing both arguments.
7. Return the raw result from the contract and any error if exists.

In summary, the Gin API serves as a middleware layer, bridging the Laravel backend with the Hyperledger Fabric network. Its primary role is to ensure that every business operation initiated in the application is translated into a validated blockchain transaction. Through defined models, routes, controllers and transaction component, the API enables structured request handling, validation, and interaction with deployed smart contracts. By centralizing timestamp generation and enforcing business rules before invoking blockchain transactions.

Furthermore, the separation between controllers and the transaction component provides clarity, maintainability and separation of concerns, ensuring that system logic remains coherent and extensible. This structure help us to maintain consistency and accountability in the application's blockchain interactions.

4.3 Chaincode Smart Contract

As previously mentioned, the blockchain utilized in this project is Hyperledger Fabric. The smart contract for Hyperledger Fabric is programmed in Chaincode. Chaincode can be implemented in various programming languages, including Go, Java, and JavaScript (including TypeScript). The primary language used is Go, and the documentation for Go Chaincode smart contracts is more comprehensive. Consequently, we have opted to program the smart contract using Go.

Based on requirements list (3.4.3) for the smart contract component of MiCoLEC, a series of functions were implemented. We will now explain two illustrative examples: a ledger write and a ledger query. The full smart contract implementation is be available in the online repository [76].

4.3.1 Ledger Write: Create Auction example

We will use the creation of a auction as example. In MiCoLEC project to create an auction the following information is needed: auction information, and parcels that will be included on the auction. Since one auction must have one or more parcels, it is necessary also to introduce the auction_has_parcel this entity is used create the relation between auction and parcels.

Before dive into logical code, we will introduce the entities first, we will start with auction entity model, represented bellow:

Code 7: Go implementation of auction model

```
type AuctionState string
const (
   AuctionClosedNoBids AuctionState = "CLOSED NO BIDS"
   AuctionClosedBids AuctionState = "CLOSED"
   AuctionOpen       AuctionState = "OPEN"
)

type Auction struct {
   ID                     string      `json:"id"`
   StartDate              time.Time   `json:"start_date"`
   EndDate                time.Time   `json:"end_date"`
   MaximumAcceptedLicitation float32  `json:"maximum_accepted_licitation,omitempty"`
   State                  AuctionState `json:"state"`
   ParticipantId          string      `json:"participant_id"`
   DocType                string      `json:"docType"`
}
```

In MiCoLEC project an auction should have the following attributes:

- **ID** - unique identifier
- **StartDate** - auction starting date.
- **EndDate** - auction expiration date.
- **MaximumAcceptedLicitation** - The max value that this auction will allow
- **State** - represent the state of the auction, this is ans enum that can take the following values: "CLOSED NO BIDS", "CLOSED" or "OPEN".
- **ParticipantId** - The unique identifier of the user that create the auction.
- **DocType** - This attribute is used in every model, and is used to identify the type of entity.

Now let's look into the parcel attributes and the model implementation of it. A parcel has a unique ID, state (Pending, Auction, Delivery, Delivered), the date that it was added to the platform, the required delivery date, address information (postal code), a bitcircle reward, weight, volume, a logistic operator ID, and an end customer ID. The translation to Chaincode Go model is the following:

Code 8: Go implementation of parcel model

```
type ParcelState string
const (
   ParcelStatePending ParcelState = "Pending"
   ParcelStateAuction ParcelState = "Auction"
   ParcelStateDelivery ParcelState = "Delivery"
   ParcelStateDelivered ParcelState = "Delivered"
)
type Parcel struct {
   ID                 string     `json:"id"`
   State              ParcelState `json:"state"`
   AddedToPlatform    time.Time  `json:"added_to_platform"`
   RequiredDeliveryDate time.Time `json:"required_delivery_date"`
   PickupPostalArea   string     `json:"pickup_postal_area"`
   DeliveryPostalArea string     `json:"delivery_postal_area"`
   NotifiedCeOption   bool       `json:"notified_ce_option"`
   BitcircleReward    int        `json:"bitcircle_reward"`
   Weight             string     `json:"weight"`
   Volumes            int        `json:"volume"`
   LogisticOperatorId int        `json:"logistic_operator_id"`
   EndCustomerId      int        `json:"end_customer_id"`
   DocType            string     `json:"docType"`
}
```

Since we already introduce the auction and parcel models the only part missing is the auction_has_parcel entity. The model is quite simple because it is just 3 properties: auction ID, parcel ID and DocType.

Code 9: Go implementation of auction_has_parcel model

```
type AuctionHasParcel struct {
   AuctionID string `json:"auction_id"`
   ParcelID string `json:"parcel_id"`
   DocType  string `json:"docType"`
}
```

Now that we have the data models set, lets look into the auction creation process on the smart contract.

Within our framework, every smart contract method follows the same implementation practices: the existence of a primary method that is responsible for the overall flow of the request.

In the example of creating an auction, the initial step involves the creation of an object representing the entity auction. Subsequently, the entity auction_has_parcel is created, and the parcel's state is subsequently updated to 'Auction'.

Code 10: Go implementation of Auction Create Main

```
func (s *SmartContract) AuctionCreateMain(stub shim.ChaincodeStubInterface, auction
    models.Auction, parcels []string) pb.Response {
  transactionError := false
  transactionId, err := s.StartTransaction(stub)
  if err != nil {
    return shim.Error(err.Error())
  }
  defer func() {
    if transactionError {
      if err := s.CloseTransaction(stub, transactionId, true); err != nil {
        fmt.Println("Error rolling back transaction:", err)
      }
    } else {
      if err := s.CloseTransaction(stub, transactionId, false); err != nil {
        fmt.Println("Error committing transaction:", err)
      }
    }
  }()
  err = s.AuctionCreate(stub, auction)
  if err != nil {
    transactionError = true
    return shim.Error(err.Error())
  }
  err = validateCreateAuctionHasParcel(parcels)
  if err != nil {
    transactionError = true
    return shim.Error(err.Error())
  }
  for _, parcelId := range parcels {
    err = s.AuctionHasParcelCreate(stub, auction.ID, parcelId)
    if err != nil {
      transactionError = true
      return shim.Error(err.Error())
    }
    err = s.ParcelAuctionStartUpdateState(stub, parcelId)
    if err != nil {
      transactionError = true
      return shim.Error(err.Error())
    }
  }
```

```
   responseItem := struct {
      Auction models.Auction `json:"auction"`
   }{}
   responseItem.Auction.ID = auction.ID
   bytes, err := json.Marshal(responseItem.Auction.ID)
   if err != nil {
      transactionError = true
      return shim.Error(err.Error())
   }
   return shim.Success(bytes)
}
```

It is possible to verify that "AuctionCreate," "AuctionHasParcelCreate," and "ParcelAuctionStartUpdateState" are the primary actions involved in the auction creation process. Now, lets dive in the creation process of each of these functions. We will begin by presenting the "AuctionCreate" function.

Code 11: Go implementation of Auction Create

```
func (s *SmartContract) AuctionCreate(stub shim.ChaincodeStubInterface, auction
    models.Auction) error {
   var compositeKey string
   compositeKey, err := s.CreateCompositeKey(stub, EntityAuction,
       []string{fmt.Sprint(auction.ID)})
   if err != nil {
      return err
   }
   err = validateAuctionStart(auction)
   if err != nil {
      return err
   }
   recordExists, err := s.EntityRecordExists(stub, compositeKey)
   if err != nil {
      return err
   } else if recordExists {
      return fmt.Errorf("record already exists")
   }
   auction.DocType = string(EntityAuction)
   bytes, err := json.Marshal(auction)
   if err != nil {
      return err
   }
   _, err = s.UpsertEntityRecord(stub, compositeKey, bytes)
   if err != nil {
      return err
   }
   return nil
}
```

The initial step in the auction create process is the creation of a composite key. Following the creation of the composite key, we validate the auction information, in order to validate, a function is implemented to verify the validity of the auction properties for creation. After properties validation we check if this is a duplicated auction, if not we proceed to auction insert on the ledger, otherwise we send an error message, saying that the record already exists on the ledger.

Given that "AuctionHasParcelCreate" is a creation, so the process of creating a record is nearly identical. Therefore, we will now proceed to the update of the parcel state in "ParcelAuctionStartUpdateState." The primary objective of this function is to update the parcel state to "Auction" state. To achieve this, we have implemented the following Chaincode:

Code 12: Go implementation of update parcel state

```
func (s *SmartContract) ParcelAuctionStartUpdateState(stub shim.ChaincodeStubInterface,
    parcelId string) error {
   compositeKey, err := s.CreateCompositeKey(stub, EntityParcel,
       []string{fmt.Sprint(parcelId)})
   if err != nil {
      return err
   }
   entity, err := s.ReadEntity(stub, compositeKey)
   if err != nil {
      return err
   }
   var parcel models.Parcel
   err = json.Unmarshal(entity, &parcel)
   if err != nil {
      return err
   }
   err = validateUpdateParcelAuctionStart(string(parcel.State))
   if err != nil {
      return err
   }
   parcel.State = models.ParcelStateAuction
   bytes, err := json.Marshal(parcel)
   if err != nil {
      return err
   }
   _, err = s.UpsertEntityRecord(stub, compositeKey, bytes)
   if err != nil {
      return err
   }
   return nil
}
```

When looking careful to the code, we can see a validation function, that at first look can not make much sense because we are only changing the parcel state to "Auction", but MiCoLEC project only allow that change if the parcel is on "Pending" state otherwise it should throw an error informing the user that this parcel is not allowed to add on the auction.

In the context of auction create, we have already covered the ledger write options, create record and update record functionalities. In the next step, we will delve into the process of query the ledger.

4.3.2 Ledger Query - Parcel Queries examples

In the previous subsection, we demonstrated how to create and update records. In this subsection, we will explore how to retrieve information from the ledger. We will present three examples, query all parcels, query a single parcel by id and query parcels by state.

The first one will go through the complete code for querying the ledger. The other two examples we will provide a concise overview of the changes made, offering valuable insights into the functionality of the automatic generation of smart contracts.

First lets look into the query all parcels examples, in order to do it we develop the following code:

Code 13: Go implementation of query all parcels

```
func (s *SmartContract) ParcelQueryAll(stub shim.ChaincodeStubInterface) pb.Response {
   queryString := `
      {"selector":
         {"docType":"PARCEL"}
      }`
   fmt.Println(queryString)
   resultsIterator, err := stub.GetQueryResult(queryString)
   if err != nil {
      return shim.Error(err.Error())
   }
   defer resultsIterator.Close()
   if !resultsIterator.HasNext() {
      return shim.Success(nil)
   }
   var results []models.Parcel
   for resultsIterator.HasNext() {
      queryResponse, err := resultsIterator.Next()
      if err != nil {
         return shim.Error(err.Error())
      }
      record := models.Parcel{}
      if err := json.Unmarshal(queryResponse.Value, &record); err != nil {
```

```
            return shim.Error(fmt.Sprintf("Error unmarshaling record: %s", err.Error()))
        }
        results = append(results, record)
    }
    jsonResponse, err := json.Marshal(results)
    if err != nil {
        return shim.Error(fmt.Sprintf("Error marshaling JSON: %s", err.Error()))
    }
    return shim.Success(jsonResponse)
}
```

This code is responsible for retrieving all parcels from the ledger. The filtering of this entity is performed using the "DocType" property, which is present in all models of the smart contract. The filtering is done in this block of code:

Code 14: Filtering data by DocType on ledger quering

```
queryString := `
    {"selector":
        {"docType":"PARCEL"}
    }`
```

Now if we want to filter by parcel id like is pretend on the second example, we just need to change the query string, to take the id into consideration. So the only change done in the code is this one:

Code 15: Filtering data by id

```
queryString := fmt.Sprintf(`
{"selector":
    {"docType":"%s","id": "%s"}
}`, EntityParcel, id)
```

In this manner, we will retrieve only the parcel with the ID that is received as a parameter from the request. This lead us to the third example which is to filter by state, in order to do so we just need to change the query string again.

Code 16: Filtering data by state

```
queryString := fmt.Sprintf(`
{"selector":
    {"docType":"%s","state": "%s"}
}`, EntityParcel, state)
```

In order to change and get different filters we just need to change the property values, and in other examples where the DocType is not a parcel, we can also change, for example all open auction the query string will be something like this:

Code 17: Filtering data on ledger quering

```
queryString := fmt.Sprintf(‘
{"selector":
   {"docType":"%s","state": "%s"}
}‘, EntityAuction, state)
```

Having acquired the fundamental knowledge of data reading and writing on the ledger, we can now proceed to the next section with the automatic generation of smart contracts.

4.4 DEMO Automatic Smart Contract Generation

Information system implementation aims to improve organizational performance, but success depends on factors such as system capabilities and organizational characteristics. DEMO utilizes the PSI theory of Enterprise Ontology to examine an organization's essence and creates interconnected models and diagrams for an accurate representation of reality [62]. We propose a new approach to smart contract development from DEMO models, with the expectation of making it less error-prone and more efficient. With a recent proposal of DEMO's action rule specification (ARS) language, it is possible to provide a more comprehensive and formal specification of logical and mathematical expressions, as well as flow control of business rules [62]. This level of precision is crucial for smart contracts, where even the slightest error could result in costly consequences. Moreover, our approach caters to a wider audience by allowing the specification of rules in a visual programming language. This not only makes it easier for non-technically savvy collaborators to participate in the process, but also enhances the overall effectiveness of the development cycle thanks to the combination of formal language and visual programming. The new ARS language elements we propose in the context of this dissertation offer a unique advantage by supporting constructs and rules with references to features such as smart contracts, documents, and templates. This, in turn, has a significant impact on the essential models built on top of DEMO's Action Model (AM), bringing their implementation closer to a real-world enterprise information system (EIS).

In order to facilitate from now on we will refer to our method as DASCG (DEMO Automatic Smart Contract Generation).

To accomplish DASCG, we extend the current DISME evolution of DEMO based EBNF grammar for the AM (Table 4) to facilitate more comprehensive specification of business rules, flow control and data store and retrieval in the context of a blockchain ledger, and enable clear visual specification.

By incorporating visual components to specify, for example, business rules and ledger queries, we aim to overcome the technical barriers faced by non-technical stakeholders, thereby leveraging a more effective collaborative smart contract development. On the other hand, we have smart contracts that are created based on thorough models that can be validated by non technical users and do not require manual coding. This combination of formal modeling and visual programming aims to improve the accessibility and efficiency of the smart contract development cycle.

Table 4: Blockchain Component Action Rule's EBNF Syntax

when	WHEN transaction_type IS | HAS_BEEN transaction_state { action}-
local_endpoint _parameter	STRING property documentation_description parameter_example_value validation_condition [MANDATORY] NOTE: the parameter name that will be used to specify the parameter is independent of the internal property that is linked to it, and therefore doesn't have to be the equal to its name.
local_endpoint_request _body	{local_endpoint_parameter | local_endpoint_parameter_set}- NOTE: defines the structure of the API Call's request_body, defining its parameters and/or parameter_sets.
local_endpoint _parameter_set	STRING documentation_description {local_endpoint_parameter}- NOTE: After specifying the set_name, it is needed to specify the parameters that are to be inserted into it.
local_endpoint_response	local_endpoint_success_response local_endpoint_error_response NOTE: must have a behaviour selected for each one. One defined in case the API call is carried out successfully, and one in case it encounters any errors in the process.

local_endpoint_success_response	local_endpoint_success_response_affected_object | local_endpoint_success_response_queried_object | STRING NOTE: When successful, it can return the created/queried/updated/deleted object or simply a custom success message. NOTE: API Call CRUD operation that will be carried influences the dropdown choices that will be present in the 'response' block. For 'create', we will have the 'created object', for 'read' we will have the local_endpoint_success_response_queried_object, for 'update' we will have the 'updated object', and for 'delete' we will have the 'deleted object'. Except for the 'read' crud operation, we have the option to output a custom success message instead of the object.
local_endpoint_success_response_affected_object	STRING {response_property | response_set}- NOTE: defines the returning object with the object's name and the properties/property_sets that should be returned inside it.
response_property	STRING property NOTE: has to be a property belonging to the object that was created/updated/queried/deleted
response_set	STRING {response_property}- NOTE: defines the set name and the properties that should be included in this response set.
local_endpoint_success_response_queried_object	STRING NOTE: the variable names defined in the 'query records' blocks will appear here in a dropdown and the user must select which ones to include in the api call's response. Will return the objects from the selected queries.
local_endpoint_error_response	STRING NOTE: error message to be returned.
action	causal_link | assign_expression | user_input | edit_entity_instance | user_output | produce_doc | if | while | for_each | **post** | **get** | **create_record** | **query_records** | **update_record** | **delete_record**
assign_expression	property "=" (term | property_value) **[BLOCKCHAIN_EXECUTION]**
property	STRING NOTE: has to be an existent property specified in table property
validation_condition	[NOT] validation_condition_type [EXTRA_FIELD_1 [EXTRA_FIELD_2]] [user_output] NOTE: The not, extrafield1 and extrafield2 fields will appear depending on the validation_condition_type chosen. EXTRAFIELD1 and EXTRAFIELD2 examples are the min and max fields if the validation_condition_type is "Belongs Range".

validation_condition_type	REQUIRED | IS_NUMBER | IS_INTEGER | EQUAL_TO | MAX_WORD_LENGTH | LESS_EQUAL | HIGHER_EQUAL | HIGHER_THAN | LESS_THAN | MIN_LENGTH | BELONG_SRANGE | MAX_LENGTH | MIN_WORD_LENGTH | HAS_CHARACTER | REG_EXPRESSION | HAS_WORD | IS_EMAIL | IS_URL | CUSTOM_VALIDATION
compute_expression	term {compute_operator term}-
compute_operator	“+”| “-”|“*”|“/” NOTE: for now simple mathematical operators, later maybe more...
if	IF condition THEN { action | rollback_transaction } - [ELSE { action | rollback_transaction } -]
condition	(ISTRUE | NOT evaluated_expression | condition) | (AND|OR { evaluated_expression | condition }-) NOTE: if condition is of type ISTRUE engine will evaluate the expression; if condition is of type NOT, engine will evaluate either the expression or condition; if it is of type AND or OR it will accordingly evaluate the specified expressions/conditions
evaluated_expression	comp_evaluated_expression | user_evaluated_expression
comp_evaluated_expression	term logical_operator (term | property_value)
logical_operator	“<” | “>” | “==” | “!=”
property_value	STRING NOTE: must be a possible value of a property with 2 cases: 1) value_type is enum and one can select all allowed values for the selected property 2) value-type is prop_ref where possible values to select will be the values associated with the property fk_property_id.
term	constant | value | property | query | compute_expression | produce_doc
constant	STRING NOTE: Can be a constant defined on the system or can be a new specification of a constant. In the latter case, the name, value and value type of the constant to be created must also be specified.
value	value_type STRING NOTE: free value inserted when editing the action rule. Must also specify the value type when specifying the new value.
value_type	TEXT | INTEGER_NUMBER | REAL_NUMBER | BOOLEAN | ENUM | DATE | TIME
for_each	FOR_EACH set {action}-
set	STRING NOTE: set of elements. Has to be a name of a ‘parameter set’ defined in the action rule’s ‘request body’ input.

create_record	entity_type {matching_property}- [ALLOW_DUPLICATES] [BLOCKCHAIN_EXECUTION] NOTE: Allows the creation of an entity in the system. 'Allow duplicates' is a checkbox that will/won't allow the insertion of duplicate records of entities of the entity type selected.
entity_type	STRING NOTE: has to be an existent entity_type specified in table ent_type
update_record	entity_type matching_id_property matching_property- [BLOCKCHAIN_EXECUTION] NOTE: properties to be updated in the entity are the matching_properties inserted. The matching_id_property is to know which entity to update.
delete_record	entity_type matching_id_property [BLOCKCHAIN_EXECUTION] NOTE: will search for the selected entity type's entity with the received api call id and will delete that entity.
rollback_transaction	STRING NOTE: does a rollback on the current action rule's transaction being executed and returns the custom error message defined. NOTE: For now, rollback_transaction is only used in blockchain action rules, but in the future will be reused for general action rules.

In the next sections, we will discuss the process of converting DEMO Action Rule Specifications to Hyperledger Chaincode Go. Due to the complex nature of the implementation code, in some examples we use pseudocode to facilitate the comprehension of the example. The actual implementation code is available in the online repository [76], enabling the interested reader to thoroughly analyze the technical details.

To address the research question that aims to validate if a specification of the DEMO language, based on an expansion of it, is capable of generating Hyperledger smart contracts, to achieve it we used reverse engineering.

First step was implementing the smart contract code as required by MiCoLEC project, and, afterwards, the the respective needed DEMO language elements for each transaction were devised, by reverse engineering to map the blockchain code to DEMO ARS. Reverse engineering involves analyzing a system to determine its components and their relationships to extract useful information. By analyzing the GO code and DEMO language specifications, it was possible to identify common elements and patterns that can be used to transpile from one language to the other. It

provided insights into how the two languages relate to each other and allowed to refine the process of generating smart contracts from DEMO specifications.

4.4.1 Essential Methods of our Framework

To streamline the transpiling of DEMO language to Chaincode implemented in GO language, several default methods have been developed. These methods are integral to all smart contracts automatically generated by DASCG

- **CreateCompositeKey**: This method creates a composite key for a specific entity in the context of a transaction in a smart contract. The composite key is created by concatenating a string that represents the entity and a list of strings that represent the entity's identifiers. The composite key is a way to efficiently index, and search records stored in the ledger of a blockchain. When the composite key is created, the blockchain indexing system stores the key in an index table. This allows queries for specific entities and associated identifiers to be executed more quickly, as the data is already indexed and can be accessed quickly by the system. The utility of this method is to provide a convenient and an efficient way to create composite keys in smart contracts facilitating the indexing of records on the blockchain. This optimizes access to stored data, allowing for better performance of the blockchain applications.

Code 18: Go implementation of CreateCompositeKey

```
func (s *SmartContract) CreateCompositeKey(stub shim.ChaincodeStubInterface,
    entity Entity, ids []string) (string, error) {
compositeKey, err := stub.CreateCompositeKey(string(entity), ids)
return compositeKey, err
}
```

- **UpsertEntityRecord**: The purpose of the "UpsertEntityRecord" method is to insert or update an entity record in a distributed ledger, which is managed by the blockchain platform on which the contract is being executed. This method is useful for decentralized applications that need to store data in a distributed ledger and ensure that the data is immutable and secure. By using a smart contract to manage access to the ledger, it is possible to ensure the integrity and validity of the data, as well as the security of the transactions.

Code 19: Go implementation of UpsertEntityRecord

```
func (s *SmartContract) UpsertEntityRecord(stub shim.ChaincodeStubInterface, key
    string, data []byte) (bool, error) {
err := stub.PutState(key, data)
```

```
if err != nil {
   return false, fmt.Errorf("failed to put data for key %s: %w", key, err)
}
return true, nil
 }
```

– **EntityRecordExists**: The method "EntityRecordExists" is a function defined in a smart contract that checks if a record with the specified key exists in the world state of the blockchain. This function can be useful in various blockchain applications, as it allows verifying if a record with a specific key already exists in the world state of the blockchain before attempting to create a new record with the same key. This helps to avoid conflicts and ensure the integrity of data stored on the blockchain.

Code 20: Go implementation of EntityRecordExists

```
 func (s *SmartContract) EntityRecordExists(stub shim.ChaincodeStubInterface, key
     string) (bool, error) {
recordJSON, err := stub.GetState(key)
if err != nil {
   return false, fmt.Errorf("failed to read from world state: %v", err)
}
return recordJSON != nil, nil
 }
```

– **ReadEntity Method**: The method “ReadEntity”, shown in Code 21, as its name suggests, is designed to retrieve an asset based on its key, key is received as a parameter, before retrieving the asset, it performs error validation and checks if the record for the specified key exists. This method is advantageous because it separates the read logic and eliminates code duplication.

Code 21: Go implementation of ReadEntity

```
func (s *SmartContract) ReadEntity(stub shim.ChaincodeStubInterface, key string)
    ([]byte, error) {
  recordJSON, err := stub.GetState(key)
  if err != nil {
     return nil, fmt.Errorf("failed to read from world state: %v", err)
  }
  if recordJSON == nil {
     return nil, fmt.Errorf("the asset %s does not exist", key)
  }
  return recordJSON, nil
}
```

– **StartTransaction & CloseTransaction**: The "StartTransaction" function is used to initiate a transaction and generate a unique ID for it using the GetTxID method of the stub (a data handling object provided by the ContractAPI). The purpose of this function is to provide a starting point for the execution of the smart contract's business logic. It also helps track the transaction and identify the exact moment it started. The "CloseTransaction" function is used to finalize the transaction. If an error occurs during the execution of the transaction, the function receives a boolean value err equal to true and performs a rollback of the transaction, If there are no errors, the function emits a "tx.commit" event to indicate that the transaction was successful. The purpose of this function is to ensure that the transaction is executed safely and consistently, and that the state of the SC is updated correctly. It also helps notify other stakeholders about the result of the transaction and maintain a record of successful and failed transactions.

Code 22: Go implementation of StartTransaction

```
func (s *SmartContract) StartTransaction(stub shim.ChaincodeStubInterface)
    (string, error) {
transactionId := stub.GetTxID()
err := stub.SetEvent("tx.start", []byte(transactionId))
if err != nil {
  return "", err
}
return transactionId, nil
}
```

Code 23: Go implementation of CloseTransaction

```
func (s *SmartContract) CloseTransaction(stub shim.ChaincodeStubInterface,
    transactionId string, err bool) error {
if err {
  // Roll back the transaction
  err := stub.SetEvent("tx.rollback", []byte(transactionId))
  if err != nil {
    return err
  }
  err = stub.DelState(transactionId)
  if err != nil {
    return err
  }
  return nil
}
// Commit the transaction
comitErr := stub.SetEvent("tx.commit", []byte(transactionId))
if comitErr != nil {
  return comitErr
```

```
    }
    return nil
     }
```

4.4.2 DASCG: Entities

To develop a smart contract, firstly we need to extract relevant information from the DEMO specification. This involves identifying the entities that participate in the smart contract based on the Entity specification in DEMO language, and converting them into Go structs that will be used as models in the Chaincode.

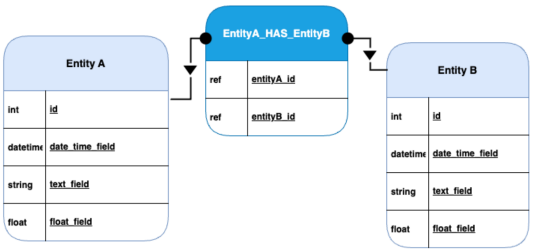


Figure. 9: General Fact Model

To store assets in a struct on the ledger, it is necessary to translate the DEMO Fact Model into Chaincode in Go. This translation can be achieved through straightforward mapping, with some adjustments as necessary.

Given that the DEMO Fact Model is specific about entities and their attributes, the translation can be conducted in the following manner.

Using the general example present in Figure 9, we will demonstrate, shown in Code 24, how this mapping between DEMO Fact Model and Chaincode is done.

Code 24: Creating Go structs based on Fact Model

```
type EntityA struct {
    Id int `json:"id"`
    DateTimeField time.Time `json:"date_time_field"`
    TextField string `json:"text_field"`
    FloatField float32 `json:"float_field"`
    DocType string `json:"docType"`
}

type EntityA_HAS_EntityB struct {
    EntityA_Id int `json:"entity_A_id"`
```

```
    EntityB_Id int `json:"entity_B_id"`
    DocType string `json:"docType"`
}

type EntityB struct {
    Id int `json:"id"`
    DateTimeField time.Time `json:"date_time_field"`
    TextField string `json:"text_field"`
    FloatField float32 `json:"float_field"`
    DocType string `json:"docType"`
}
```

In this section, we represent all entities in a single block of code in order to facilitate the interpretation of the transpiling between DEMO Fact Model into Chaincode Go Struct, but on DASCG, each entity will have a designated file located within the "models" directory. This document will provide later a comprehensive analysis of the file structures of DASCG.

4.4.3 Ledger Storage MiCoLEC Example

In this section, we will delve into the process of transpiling DEMO Action Models to blockchain transactions encoded in SCs, encompassing multiple creations and updates. To illustrate this, we will examine the creation of an auction, shedding light on the significance of each action within the DEMO ARS. In the MiCoLEC project, creating an auction involves conducting multiple validations and establishing a relationship between a parcel and the auction itself. Specifically, this process necessitates the creation of the "Auction_Has_Parcel" entity. This entity acts as a pivot table, managing the many-to-many relationship between Auction and Parcel entities. Additionally, it is imperative to update the state of the parcel from "Pending" to "Auction", indicating that the parcel is part of an ongoing auction. Prior to making this update, a crucial validation step must be taken to ensure that the parcel's current state is indeed "Pending".

Once the entities are defined, the next step is to map all the actions specified in the DEMO specification to the corresponding Chaincode. This mapping enables implementation of a process for transpiling between DEMO and Chaincode, enabling the generation of correct smart contracts. The smart contract method name is derived from the pattern used on Action Rule (AR) in the Grammar.

We will first dissect the AR into smaller segments to gain a deeper understanding of the significance of each component within the AR, and to comprehend how it is transpiled into Chaincode.

Firstly, we will delve into fundamental programming concepts, including conditional structures and repetition mechanisms and how the transpiling happens from AR to Chaincode.

The conditional structures, help us to control the program flow, allowing to perform logic-base decisions, the definition of this structures in DEMO AR can be achieved using the block "IF-THEN-ELSE" specified on Figure 10.

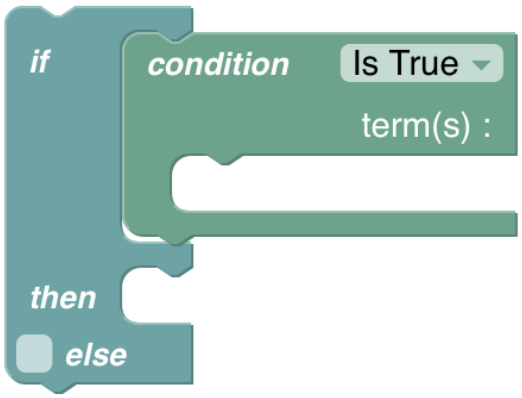


Figure. 10: DEMO Action Rule: "IF-THEN-ELSE" block.

This block defines which action should take place based on an evaluation of the condition, if the condition criteria are fulfilled, the blocks within the “then” section become valid. Otherwise, the “else” option is selected and the blocks within the "else" are executed.

To achieve the same result on Chaincode, we transpile this block present on Figure 10 into the following code.

Code 25: DEMO Action Rule "IF-THEN-ELSE" to Chaincode Go

```
if CONDITION {
    // ADD ACTION HERE
}
else{
    // ADD ACTION HERE
}
```

With this understanding, we can now better interpret the validation condition of a property. The condition is interpreted as an "IF condition THEN action [ELSE action]" grammar. However, it can be simplified further. If the condition evaluates to true, then the validation criteria doesn’t match, and an error message is generated based on the property name and condition. For example,

if we have a property of type "int", it must be greater than '0'. Therefore, if the condition is not met, an error message will be generated.

Code 26: Creating validation method to user input

```
func validatetransaction_type(entity *models.EntityName) error{
    var errorMessages []string
    if !(entity.propertyName > 0) {
        errorMessages = append(errorMessages, fmt.Sprintf("%s most be bigger than %d ",
            entity.propertyName, 0))
    }
    // Validate all properties of the entity
    (...)
    if len(errorMessages) > 0 {
        return errors.New(strings.Join(errorMessages, "\n"))
    }
    return nil
}
```

The foundation of the AR is encapsulated in the "When" block, as illustrated in Figure 11. This segment explicitly outlines various critical components: the AR name, the parameters required for the blockchain request, and the series of actions slated for execution. Additionally, it defines the response mechanism, indicating whether the action has been successfully executed or if any errors have emerged during the process.

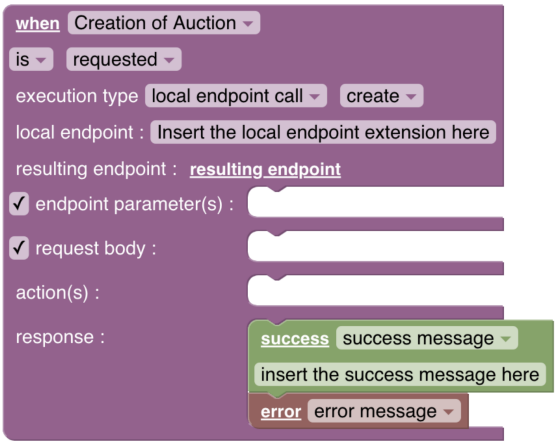
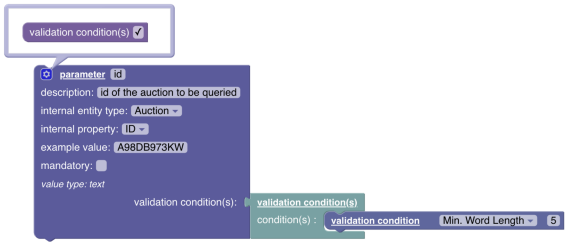


Figure. 11: DEMO Action Rule: "When" block

In Figure 11 as said previously, we have a property that handles with the parameters that our blockchain method have to receive in order to use it in the actions. We can have two types of parameters:

– The "parameter" block symbolizes a singular value, as depicted in Figure 12. The "parameter" block can have a validation clause if it is selected on the setting of this block.

– The "parameter set" represents the second type of parameter, and it denotes a list of parameters sharing the same structure. This format streamlines repetitive information by eliminating multiple insertions of the same data type, enhancing intuitiveness and efficiency. Validations can be incorporated since the “set parameter” accommodates individual “parameter” as its properties, allowing for comprehensive data integrity checks.



s

Figure. 12: DEMO Action Rule: Parameter Block with validation

With a comprehensive understanding of the “When,” “parameter,” and “set parameter” blocks, we will proceed to analyze the Chaincode generated from these blocks. To facilitate this analysis, we will draw upon the example employed in our project. The example is an auction creation, where we encounter multiple “parameter” blocks associated with the auction’s properties, as well as a “set parameter” block representing the parcels to be included in the auction. The “When” block generates the method’s structure, taking the parameters as input for the base method.

Code 27: DEMO Action Rule "WHEN" to Chaincode Go

```
func (s *SmartContract) AuctionCreateMain(stub shim.ChaincodeStubInterface, auction
    models.Auction, parcels []string) pb.Response {
   pseudoInitiateTransactionEnsureAtomicity()
   //ADD ACTIONS HERE
   (...)
   return shim.Success(nil) }
}
```

To guarantee atomicity, we make use of the "StartTransaction" and "CloseTransaction" methods, it handle potential errors during the transaction, and ensures the transaction is either rolled back if there were errors or committed if the transaction was successful.

With the foundational method established, we can delve deeper into incorporating actions within the DEMO Action Rules, specifically for executing data writes on the ledger. To facilitate this, we will introduce and discuss two blocks capable of writing data to the ledger: "create record" and "assign expression".

As represented in Figure 13, the "create record" possesses several properties. The "entity type" is the property that defines the type of asset that will be created and added to the ledger. The "allow duplicates" property that can be enabled if in a specific situation warrants the creation of duplicate records. And finally, the "property(ies)" property that assigns values to the various auction attributes. By employing the "matching" block that is responsible by link auction properties to Action Rule parameters, that have been explained before, this ensures proper initialization and consistency in the data recorded on the ledger.

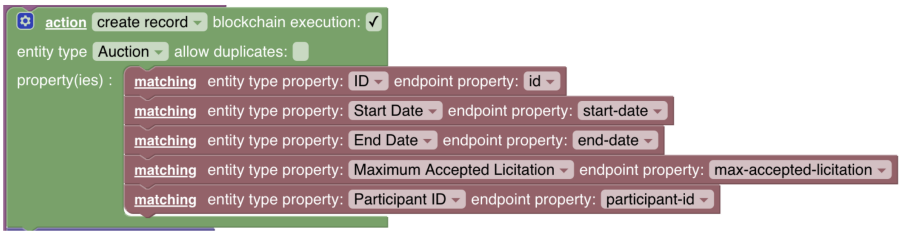


Figure. 13: DEMO Action Rule: Create Auction

Having all the necessary information that is needed in the block of creation, we can check how it will be translated to Chaincode.

Code 28: Pseudo Code of Creating Auction

```
func (s *SmartContract) AuctionCreate(stub shim.ChaincodeStubInterface, auction
    models.Auction) error {
    pseudoCreateCompositeKey()
    pseudoValidateAuction()
    pseudoCheckDuplicates()
    pseudoSetDocType()

    bytes, err := json.Marshal(auction)
    if err != nil {
        return err
    }
    _, err = s.UpsertEntityRecord(stub, compositeKey, bytes)
    if err != nil {
        return err
    }
    return nil
```

```
}
```

The auction is notably passed as an argument into the “AuctionCreate” function, and there is a uniform sequence of actions carried out for every creation on the ledger.

1. A unique key is generated. This unique key is used to create a composite key, which facilitates the identification and retrieval of the record.
2. The properties of the entity are validated. This validation process is generated automatically, based on any validations specified in the “parameter” component.
3. Verify if duplicate values are permissible for this particular record, if not the "EntityRecordExists" from essential methods is invoked, in order to check whether it is duplicated or not.
4. A document type (docType) property is assigned to the entity, which aids in the categorization and querying of ledger records.
5. Lastly, the entity, along with all its validated properties and the assigned docType, is securely stored on the ledger.

The role remaining task is to invoke the function created within the main function, as previously demonstrated in Code 27. This function will be incorporated into the “AuctionCreateMain” function. The Code 29 illustrates the “AuctionCreateMain” function once the action has been implemented.

Code 29: Pseudo Code of Creating Auction

```
func (s *SmartContract) AuctionCreateMain(stub shim.ChaincodeStubInterface, auction
    models.Auction, parcels []string) pb.Response {
   pseudoInitiateTransactionEnsureAtomicity()
   // Create Auction Action
   err = s.AuctionCreate(stub, auction)
   if err != nil {
       transactionError = true
       return shim.Error(err.Error())
   }
   (...)
   return shim.Success(nil) }
}
```

To create the records for the entity that is responsible for the management of many-to-many relation between auction and parcel, we need to add a new component to it the “for each”block,

"for each" is a repetition structure enables the execution of a block of code multiple times. These structures are particularly useful for processing lists or arrays. They allow the programmer to avoid code duplication and to manage dynamic information efficiently. The "for each" block receives a set of parcels in this case and creates for each parcel a record of type “Auction Has Parcel”, as showed in Figure 14.

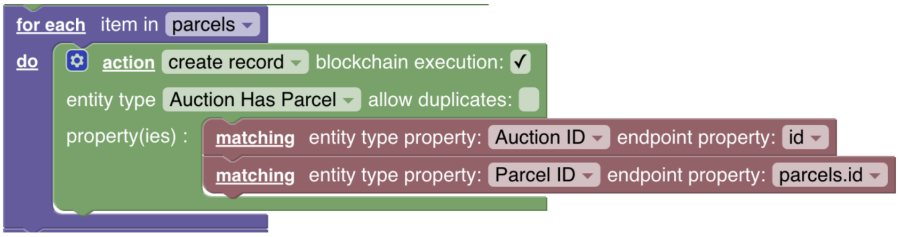


Figure. 14: DEMO Action Rule: Create Auction Has Parcel

The process of creating an "Auction" entity and an "Auction Has Parcel" entity are identical, and the function generated will be similar, they only differ on the “AuctionCreateMain” method, since "auction has parcel" is encapsulated by a "for each" block. This block accepts a set of parcels, and for each parcel, it creates a record of the type “Auction Has Parcel”. This process is visually depicted in Figure14, illustrating how the “for each” block facilitates the efficient creation of multiple relational records between auctions and parcels.

Code 30: Pseudo Code of Creating Auction Main Method with create auction has parcel added

```
func (s *SmartContract) AuctionCreateMain(stub shim.ChaincodeStubInterface, auction
    models.Auction, parcels []string) pb.Response {
    pseudoInitiateTransactionEnsureAtomicity()
    // Create Auction Action
    err = s.AuctionCreate(stub, auction)
    if err != nil {
        transactionError = true
        return shim.Error(err.Error())
    }

    // Create Auction Has Parcel
    for _, parcelID := range parcels {
        err = s.AuctionHasParcelCreate(stub, auction.ID, parcelID)
        if err != nil {
            transactionError = true
            return shim.Error(err.Error())
        }
    }
```

```
    return shim.Success(nil) }
}
```

Until now, we have explored the process of creating new assets. Now, we will focus on updating data. To accomplish this, we will utilize the "assign expression" component, example in Figure15. This component possesses several crucial properties that will facilitate our data update process.

The "Scope" property specifies the type of object to be updated. The "ent type" specifies the entity type to be updated. To define which property of the entity will be updated, we use the "property" to specify the specific property to be modified during the update operation.

On the right side of the equal sign, is set the desired value to be assigned to the updated property. This configuration guarantees precise and intentional modifications to the record properties.

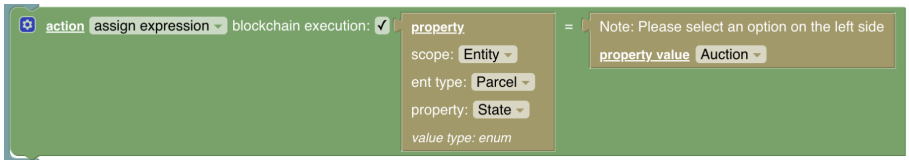


Figure. 15: DEMO Action Rule: Assign Expression

Code 31: Code of Update Parcel to Auction State

```
func (s *SmartContract) UpdateParcelStateAuction(stub shim.ChaincodeStubInterface,
    parcelId string) error {
     compositeKey, err := s.CreateCompositeKey(stub, EntityParcel,
         []string{fmt.Sprint(parcelId)})
     if err != nil {
        return err
     }
     entity, err := s.ReadEntity(stub, compositeKey)
     if err != nil {
        return err
     }

     var parcel models.Parcel
     err = json.Unmarshal(entity, &parcel)
     if err != nil {
        return err
     }

     err = validateUpdateParcelAuctionStart(string(parcel.State))
     if err != nil {
        return err
     }
```

```
    parcel.State = models.ParcelStateAuction

    bytes, err := json.Marshal(parcel)
    if err != nil {
       return err
    }

    _, err = s.UpsertEntityRecord(stub, compositeKey, bytes)
    if err != nil {
       return err
    }

    return nil
}
```

On auction creation, example e make use of the "assign expression", when we want to update the parcels "State" from "Pending" to "Auction", the “Scope” property determines the type of object to be updated; in the example depicted in Figure 16, the intention is to update an Entity record.

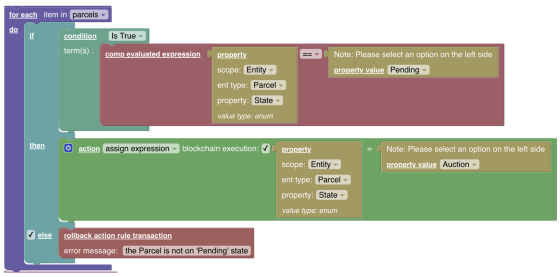


Figure. 16: DEMO Action Rule: Update parcels "State" to "Auction"

The “ent type” is specified as “Parcel”, indicating the entity type to beupdated, and the “property” is set to “State”, pinpointing the specific parcel property to be modified during the update operation. On the right side of the equal sign, the desired value to be assigned to the updated property is clearly presented. This configuration ensures precise and intentional updates to record properties.

Looking carefully at the component showed in Figure 16, we can use all the information in that component to perform the update using Chaincode, for that we will need to create the method “ParcelAuctionStartUpdateState” with the following structure:

Code 32: Pseudo Code of Updating Parcel State

```
func (s *SmartContract) ParcelAuctionStartUpdateState(stub
shim.ChaincodeStubInterface, parcelId string) error {
    pseudoCreateCompositeKey()
    pseudoReadEntity(COMPOSITE_KEY)
    pseudoValidateParcelState(PARCEL_STATE)
    parcel.State = models.State(models.ParcelStateAuction)
    bytes, err := json.Marshal(parcel)
    if err != nil {
    return err
    }
    _, err = s.UpsertEntityRecord(stub, compositeKey, bytes)
    if err != nil {
    return err
    }
    return nil
}
```

4.4.4 Query Ledger MiCoLEC Example

The DISME Dynamic Search component is a recent innovation in DISME, streamlining the process of designing queries to visualize data from various system entities. The user begins by selecting the desired entity or entities through select boxes, defining the scope of their query. Following this, they choose the specific properties they want to retrieve and designate certain properties to serve as filters in the subsequent stage. In the final step, the user applies these filters based on the properties they selected earlier. This intuitive three-step process enables users to effortlessly design data queries that suit their preferences, all while requiring no technical expertise.

To better understand the DISME Dynamic Search component, let's explore an example where a user wants to query all open auctions in the system. Initially, the user selects the "Auction" entity. Following that, in the property selection phase, the "state" property is chosen as a filter. Finally, the user specifies a filter condition, setting the "state" to "OPEN". After saving these preferences, a corresponding method is auto-generated in the smart contract, adhering to a predefined pattern to ensure consistent and accurate data retrieval.

From the example provided earlier, and based on the information supplied by the user, we can derive the query: {"selector": {"docType": "AUCTION", "state": "OPEN"}}. Utilizing this data, we can translate the user-created input from the DISME Dynamic Search into the corresponding Chaincode, ensuring seamless interaction and data retrieval from the blockchain ledger.

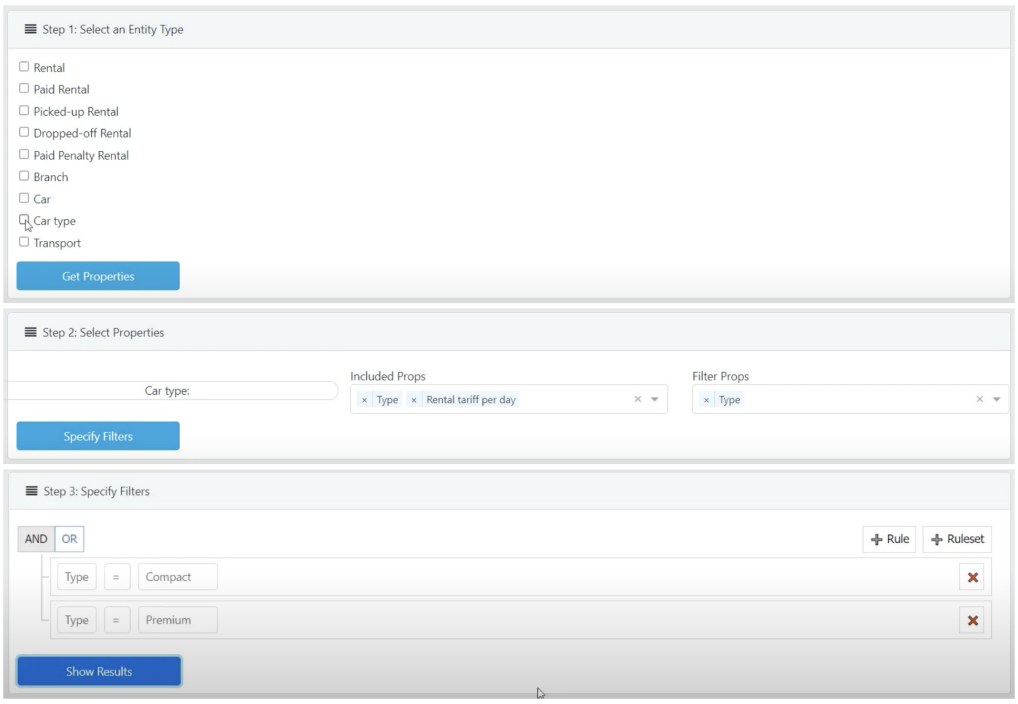


Figure. 17: DISME Dynamic Search interface example

Code 33: Query Open Auction

```
queryString := `
  {"selector":
     "{\"docType\":\"AUCTION\", \"state\":\"OPEN\"}
  "}`
resultsIterator, err := stub.GetQueryResult(queryString)
```

As data accumulates within the ledger, retrieval operations can become increasingly time-consuming. To mitigate this latency, Hyperledger Fabric introduces the concept of the "State Database", the world state maintains the current attribute values of a business object as a distinct ledger status. This is beneficial since applications typically need the present value of an object. Traversing the entire blockchain to determine an object's current value would be inefficient, instead we can directly retrieve it from the "State Database". While the "State Database" provides a structured snapshot of the ledger's information, search operations can still experience latency, as data continues to amass. To counteract this, custom indexes are tailored for each distinct query executed on the ledger. Specifically, for the "Query Open Auctions" operation, the following index was automatically generated:

Code 34: Designing an Index to Access Open Auctions

```
{
    "index": {"fields": [ "docType", "state" ]},
```

```
    "name": "auctionByStateIndex",
    "type": "json"
}
```

This analysis demonstrates how data queries can be designed in the DISME Dynamic Search component, which can subsequently be seamlessly converted into Chaincode. Additionally, performance considerations were discussed in the context of ledger searches. Enhancements were made by adding indexes to the Hyperledger's "State Database", optimizing the search queries designed for SCs.

4.5 Structure of Generated SC Files

Within the goal to automate SC generation, it was necessary to conceive an intuitive method to structure the output files resulting from the transformation of DEMO AR to Chaincode. For the proposed structure, represented on Figure 18, each AR will generate a corresponding file within the "Chaincode" directory. Simultaneously, individual entities will have their data models encapsulated in distinct files located in the "Models" directory.

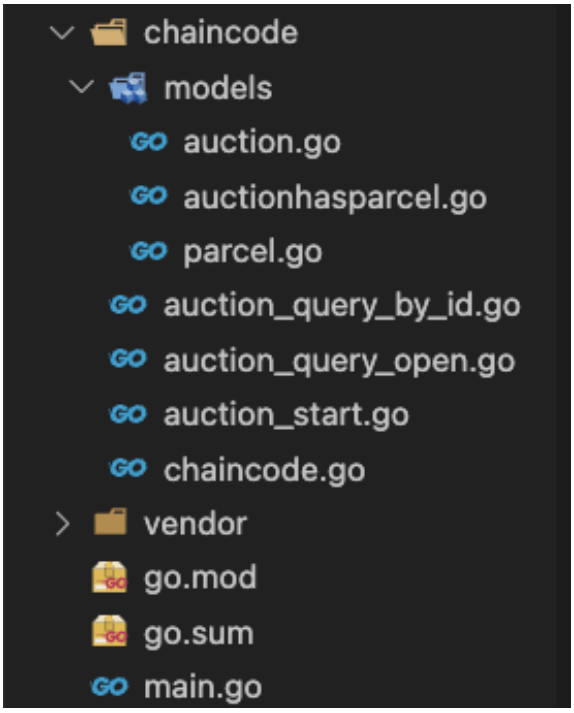


Figure. 18: Structure of Generated Files

This methodology is employed because users create each DEMO AR individually. If an error occurs or a modification is necessary in a specific AR, it is more straightforward to pinpoint

and remove the generated file associated with that particular AR. Subsequently, a new file can be created based on the alterations made to the AR. This process improves the ease of error correction and adjustment, leading to a more efficient and user-friendly experience.

5 Results, Evaluation and Discussion

This chapter summarizes the results obtained from the implementation (5.1). It evaluates the correctness of the generated smart contracts (5.2), their execution within the MiCoLEC context (5.3), and the maintainability of the generated file structure (5.4). The discussion (5.6) relates these results to the research objectives defined earlier, compares them with related work, and highlights limitations and future directions (5.7).

5.1 Overview of Results

The implementation in this work produced a proof of concept on how transpile DEMO models into Hyperledger Fabric smart contracts in order to produce automatic generation of smart contracts. The proposed approach was implemented in the MiCoLEC project, where both manually developed and transpiled smart contracts were integrated into the blockchain.

The results can be grouped into three categories:

1. Correctness of the transpilation method: validation that DEMO Action Rules were consistently mapped to executable Chaincode in Go.
2. Execution of business operations: demonstration that created smart contracts enabled essential parcel management operations within the collaborative micro-hub.
3. File structure and maintainability: assessment of how the organized code generation supported readability and potential extension.

These results were compared to the objectives in Chapter 1, reducing barriers for smart contract development and providing a model-driven approach integrating enterprise modeling with blockchain infrastructures.

5.2 Correctness of Generated Smart Contracts

The first set of results concerned the syntactic and semantic correctness of the generated smart contracts. The grammar specification introduced in Chapter 4 allowed the representation of action rules with conditional logic, parameter validation, and state updates. Transpilation of these structures into Go functions was systematically verified through tests and execution in the Hyperledger Fabric.

Examples such as Create Auction, Auction Has Parcel, and Update Parcel State confirmed that the Chaincode reflected the intended organizational rules. The execution traces indicated that the generated methods correctly wrote data to the ledger, retrieved stored entities, and enforced parameter constraints. The validation demonstrated that the mapping between DEMO's Action Model and the Chaincode was consistent.

The correctness of the generated smart contracts was verified through internal and external validations. Internal testing focused on functional behavior and code consistency, while external validation confirmed the proper execution of contracts in realistic MiCoLEC operational scenarios.

While correctness was confirmed for the tested cases, it is important to note that the evaluation was limited to the scope of the MiCoLEC use cases. More complex models with nested rule dependencies were not fully exercised in this proof of concept.

5.3 Execution within the MiCoLEC Context

The integration of generated contracts with the MiCoLEC platform confirmed the applicability of the approach in a realistic logistics scenario. The platform required operations such as parcel registration, assignment of deliveries, execution of auctions, and validation of circular economy parcels. These operations were encoded as DEMO Action Rules and subsequently mapped into Chaincode.

Execution in Hyperledger Fabric showed that the contracts were able to support collaborative interactions between logistics operators, couriers, and circular economy operators. For example, queries such as Auction By Id and Query Open Auctions returned consistent results from the distributed ledger.

The experiment further validated the suitability of Hyperledger Fabric for consortium environments, such as MiCoLEC. The permissioned nature of the platform enabled controlled participation by different stakeholders while maintaining transparency of operations.

5.4 File Structure and Maintainability

Another relevant result was the definition of a structured output for the generated files. The organization into directories for entities, rules, and auxiliary functions facilitated navigation and future extension of the codebase. From a maintainability perspective, this structure enabled developers to locate the generated artifacts corresponding to specific DEMO elements.

This result addresses one of the recurring issues in smart contract development, namely the difficulty of maintaining large codebases where business logic and technical details are interconnected. By preserving the separation between models and generated code, the approach contributes to long-term sustainability of blockchain-based solutions.

5.5 Discussion of Objectives and Research Questions

The results obtained respond to the research objectives stated in Chapter 1.

- Reduction of barriers: The translation from visual DEMO models to executable chaincode mitigates the dependency on deep technical knowledge of blockchain programming. This aligns with the objective of making smart contract development more accessible.

- Enterprise integration: By grounding contracts in the DEMO methodology, the generated artifacts preserve a direct relation to organizational semantics, which supports interoperability across enterprises.

- Automation: the defined mapping between DEMO Action Rules and executable Chaincode demonstrates that it is possible to generate automatic smart contract, leading to reducing manual coding effort.

This dissertation successfully addressed the research questions defined in Chapter 1. Concerning RQ1, the work demonstrated that it is possible to automatically generate correct and executable smart contracts from DEMO Action Rules through the definition of an extended EBNF grammar and a corresponding mapping method to Hyperledger Fabric Chaincode. The implementation and validation within the MiCoLEC context confirmed the correctness of the mapped contracts. Concerning RQ2, the results showed that the proposed model-driven approach reduces technical barriers in smart contract development by providing a structured generation process that bridges detailed graphical organizational models and blockchain implementation. Contributing to greater accessibility, transparency, and maintainability in collaborative logistics scenarios.

Nevertheless, some limitations remain. The approach requires the definition of DEMO models with a sufficient level of detail, which presupposes expertise in enterprise modeling and training is needed, but, nevertheless, it is easier for business users to read models than code. Formal evaluation comparing both approaches was not the focus of this work and is a line of future work. Furthermore,

the current grammar supports a restricted subset of constructs, which limits expressiveness. There are open ends in the grammar to be extended in future work also.

5.6 Comparison with Related Work

When compared with approaches that rely on natural language specifications or configuration-based templates, the method presented here provides a more formal foundation by relying on enterprise engineering principles. Existing works in automatic contract generation often emphasize usability at the expense of semantic rigor. The DEMO-based method offers a balance by ensuring that the generated contracts correspond to formally validated DEMO action rules specifications.

In relation to other blockchain oriented frameworks explored in the literature, the presented solution distinguishes itself by addressing not only code generation, but also the organization of generated artifacts for maintainability. This complements earlier contributions that focus primarily on runtime execution.

The integration into MiCoLEC further extends the contribution by validating the approach in a sectorial context (logistics and circular economy), whereas much of the related work remains at the level of abstract prototypes.

5.7 Limitations and Future Considerations

The identified limitations are as follows:

1. Scope of evaluation: The proof of concept was limited to MiCoLEC use cases. Broader validation across domains is needed.
2. Expressiveness of grammar: Although sufficient for the tested action rules, the grammar does not yet support the full range of DEMO constructs that need to be adapted to the context of blockchain use.
3. Performance analysis: The evaluation did not include systematic benchmarks of execution time or scalability in larger networks.

Future work may extend the grammar, generalize the transpilation process to other blockchain platforms, and evaluate the approach under realistic load conditions.

6 Conclusion

This chapter concludes the dissertation. It summarizes the contributions of the work, highlights the main limitations, and suggests directions for further research. The chapter reflects on how the proposed approach addresses the problem defined in Chapter 1 and situates the results within the context of the MiCoLEC project.

This dissertation addressed the problem of reducing the complexity of smart contract development by proposing a model-driven approach based on the DEMO methodology and targeting the Hyperledger Fabric platform. The work emerged in the context of the MiCoLEC project, whose requirements highlighted the need for trust, transparency, and automation in collaborative logistics micro-hubs.

The main research contribution, that resulted in the publication of two conference papers, was the definition of a grammar and transpilation method that enable the automatic generation of smart contracts from DEMO Action Rules. The solution also included the organization of the generated files in a structured format to facilitate maintenance and integration with application backends. Through a proof of concept, it was demonstrated that organizational models could be directly transformed into executable blockchain code, supporting operations such as parcel management, auctions creation and bids.

The results presented in this study provide comprehensive answers to the defined research questions, as previously discussed in detail in the results section. Confirming that the approach is feasible and capable of aligning organizational semantics with blockchain implementation. The evaluation showed that generated contracts were correct, executed the expected operations in Hyperledger Fabric, and were applicable in the logistics use case of MiCoLEC. This establishes a path towards lowering the barriers to smart contract adoption by reducing dependence on specialized technical expertise.

However, the study also revealed limitations. The grammar currently supports only a subset of DEMO constructs, restricting the expressiveness of generated contracts. The evaluation was limited to the MiCoLEC scenario, without extensive performance measurements or cross-domain validation. In addition, while automation reduces coding effort, the accuracy of the output depends on the quality and completeness of the input models, which presupposes expertise in enterprise modeling.

Future research should address these limitations by extending the grammar, exploring the integration with other blockchain platforms beyond Hyperledger Fabric, and conducting systematic performance and scalability evaluations. Further work could also investigate user-centered tools that allow non-technical stakeholders to model action rules and directly observe their transformation into smart contracts. Another future work could also be tried is the usage o AI, like Large or Small Language Models in order to interpret textual specifications or narrative specifications of contracts that need to be supported in blockchain and convert them into DEMO models whih can then lead to the automatic generation of smart contract code in an even more streamlined manner. In fact, a PhD thesis proposal will soon be submitted to continue current work in this line.

In conclusion, this dissertation demonstrates that combining enterprise modeling with blockchain technology provides a viable strategy to improve transparency, trust, and interoperability in collaborative environments. The contributions offer a foundation for the development of low-code or no-code platforms that integrate organizational design with distributed ledger technology, fostering broader adoption of smart contracts in logistics and other sectors.

References

[1] "Microhubs as the future of last-mile logistics - EU Urban Mobility Observatory." [Online]. Available: https://urban-mobility-observatory.transport.ec.europa.eu/news-events/news/microhubs-future-last-mile-logistics-2025-04-14_en

[2] "Circular economy strategy - Research and innovation - European Commission." [Online]. Available: https://research-and-innovation.ec.europa.eu/research-area/environment/circular-economy/circular-economy-strategy_en

[3] D. Aveiro, L. Abreu, D. Pinto, and V. Freitas, "DEMO Models Based Automatic Smart Contract Generation: A Case in Logistics Using Hyperledger," *Proceedings of the International Conference on Information Systems Development (ISD)*, Oct. 2023. [Online]. Available: https://aisel.aisnet.org/isd2014/proceedings2023/modelling/3

[4] L. Abreu, D. Aveiro, and V. Freitas, "Towards automatic smart contract generation from DEMO models: A case in logistics using Hyperledger," *Companion Proceedings of the 16th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling and the 13th Enterprise Design and Engineering Working Conference, November 28 – December 1, 2023, Vienna, Austria.*

[5] E. Tijan, S. Aksentijević, K. Ivanić, and M. Jardas, "Blockchain Technology Implementation in Logistics," *Sustainability*, vol. 11, no. 4, p. 1185, Jan. 2019, number: 4 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2071-1050/11/4/1185

[6] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," in *2017 IEEE International Congress on Big Data (BigData Congress)*, Jun. 2017, pp. 557–564. [Online]. Available: https://ieeexplore.ieee.org/document/8029379/?arnumber=8029379

[7] B. Liu, "Overview of the Basic Principles of Blockchain," in *2021 International Conference on Intelligent Computing, Automation and Applications (ICAA)*, Jun. 2021, pp. 588–593. [Online]. Available: https://ieeexplore.ieee.org/document/9653563

[8] X. Neumeyer, K. Cheng, Y. Chen, and K. Swartz, "Blockchain and sustainability: An overview of challenges and main drivers of adoption," in *2021 IEEE International Conference on Technology Management, Operations and Decisions (ICTMOD)*, Nov. 2021, pp. 1–6, iSSN: 2159-5119. [Online]. Available: https://ieeexplore.ieee.org/document/9739231

[9] N. Szabo, "Nick Szabo – The Idea of Smart Contracts." [Online]. Available: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html

[10] S. Islam and K. U. Apu, "Decentralized vs Centralized Database Solutions In Blockchain: Advantages, Challenges and Use Cases," *Global mainstream journal of innovation, engineering & emerging technology*, vol. 3, pp. 58–68, Aug. 2024.

[11] "What is a Peer-to-Peer Network? - Utimaco," Jan. 2020. [Online]. Available: https://utimaco.com/service/knowledge-base/blockchain/what-is-a-peer-to-peer-network

[12] "Peer to Peer Network in Blockchain in Detail." [Online]. Available: https://blocktpoint.com/blockchain/blockchain-peer-to-peer-network

[13] "Hashing in Blockchain - A Comprehensive Overview," Dec. 2023. [Online]. Available: https://rejolut.com/blog/hashing-in-blockchain/

[14] "The Role of Cryptography in Securing Blockchain Networks." [Online]. Available: https://www.openware.com/

[15] Simanta Shekhar Sarmah, "(PDF) Understanding Blockchain Technology," iSSN: 2018-0802. [Online]. Available: https://www.researchgate.net/publication/336130918_Understanding_Blockchain_Technology

[16] H. Xiong, M. Chen, C. Wu, Y. Zhao, and W. Yi, "Research on Progress of Blockchain Consensus Algorithm: A Review on Recent Progress of Blockchain Consensus Algorithms," *Future Internet*, vol. 14, no. 2, p. 47, Feb. 2022, number: 2 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1999-5903/14/2/47

[17] S. Velliangiri and P. Karthikeyan, "Blockchain Technology: Challenges and Security issues in Consensus algorithm," in *2020 International Conference on Computer Communication*

and Informatics (ICCCI), Jan. 2020, pp. 1–8, iSSN: 2329-7190. [Online]. Available: https://ieeexplore.ieee.org/document/9104132/?arnumber=9104132

[18] K. Azbeg, O. Ouchetto, S. Jai Andaloussi, and F. Laila, "An Overview of Blockchain Consensus Algorithms: Comparison, Challenges and Future Directions," Oct. 2020.

[19] H. Ozkul, E. Celiker, M. Aydos, and A. Ozsoy, "Comparison of Top 10 Well-Known Blockchain Consensus Algorithms," in *2023 16th International Conference on Information Security and Cryptology (ISCTürkiye)*, Oct. 2023, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/10336118

[20] W. Zhao, S. Yang, and X. Luo, "On Consensus in Public Blockchains," in *Proceedings of the 2019 International Conference on Blockchain Technology*, ser. ICBCT '19. New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–5. [Online]. Available: https://dl.acm.org/doi/10.1145/3320154.3320162

[21] A. C. D. Therry, R. Ardiansyah, M. Y. Pusadan, Y. Y. Joefrie, and A. A. Kasim, "The Implementation and Analysis of The Proof of Work Consensus in Blockchain," *Advance Sustainable Science, Engineering and Technology*, vol. 6, no. 1, p. 02401018, Jan. 2024. [Online]. Available: http://journal.upgris.ac.id/index.php/asset/article/view/17878

[22] P. D'Arco, Z. E. Ansaroudi, and F. Mogavero, "Multi-stage Proof-of-Works: Properties and vulnerabilities," *Theoretical Computer Science*, vol. 976, p. 114108, Oct. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397523004218

[23] S. Singh and S. Chakraverty, "Implementation of Proof-of-Work using Ganache," in *2022 IEEE Conference on Interdisciplinary Approaches in Technology and Management for Social Innovation (IATMSI)*, Dec. 2022, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/10119271

[24] I. Malakhov, "Analysis of the Transaction Confirmation Process and Fairness in Proof-of-Work Blockchains," *SIGMETRICS Perform. Eval. Rev.*, vol. 52, no. 3, pp. 27–30, Jan. 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3712170.3712180

[25] Y. Chen, L. Tian, L. Yang, L. Zhang, and Y. Fan, "Defects analysis of blockchain PoW consensus protocol," in *Third International Conference on Computer Science and*

Communication Technology (ICCSCT 2022), vol. 12506. SPIE, Dec. 2022, pp. 425–430. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/12506/125061T/Defects-analysis-of-blockchain-PoW-consensus-protocol/10.1117/12.2662213.full

[26] S. Fahim, S. K. Rahman, and S. Mahmood, "Blockchain: A Comparative Study of Consensus Algorithms PoW, PoS, PoA, PoV," *International Journal of Mathematical Sciences and Computing*, vol. 9, no. 3, p. 46. [Online]. Available: https://www.mecs-press.org/ijmsc/ijmsc-v9-n3/v9n3-4.html

[27] C. T. Nguyen, D. T. Hoang, D. N. Nguyen, D. Niyato, H. T. Nguyen, and E. Dutkiewicz, "Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities," *IEEE Access*, vol. 7, pp. 85 727–85 745, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8746079

[28] S. Lin, "Proof of Work vs. Proof of Stake in Cryptocurrency," *Highlights in Science, Engineering and Technology*, vol. 39, pp. 953–961, Apr. 2023. [Online]. Available: https://drpress.org/ojs/index.php/HSET/article/view/6683

[29] M. Pineda, D. Jabba, W. Nieto-Bernal, and A. Pérez, "Sustainable Consensus Algorithms Applied to Blockchain: A Systematic Literature Review," *Sustainability*, vol. 16, no. 23, p. 10552, Jan. 2024, number: 23 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2071-1050/16/23/10552

[30] L. Li, "Mitigating Challenges in Ethereum's Proof-of-Stake Consensus: Evaluating the Impact of EigenLayer and Lido," Dec. 2024, arXiv:2410.23422 [cs]. [Online]. Available: http://arxiv.org/abs/2410.23422

[31] W. Li, X. Deng, J. Liu, Z. Yu, and X. Lou, "Delegated Proof of Stake Consensus Mechanism Based on Community Discovery and Credit Incentive," *Entropy*, vol. 25, no. 9, p. 1320, Sep. 2023, number: 9 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/1099-4300/25/9/1320

[32] J. Kim, S. Oh, Y. Kim, and H. Kim, "Improving Voting of Block Producers for Delegated Proof-of-Stake with Quadratic Delegate," in *2023 International Conference on*

Platform Technology and Service (PlatCon), Aug. 2023, pp. 13–17, iSSN: 2766-4198. [Online]. Available: https://ieeexplore.ieee.org/document/10255193

[33] W. Li, C. Feng, L. Zhang, H. Xu, B. Cao, and M. A. Imran, "A Scalable Multi-Layer PBFT Consensus for Blockchain," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1146–1160, May 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9279277

[34] N. Mohanan, "Comparative study of blockchain consensus algorithms in Cryptocurrency – IJSREM." [Online]. Available: https://ijsrem.com/download/comparative-study-of-blockchain-consensus-algorithms-in-cryptocurrency/

[35] B. Yang, "Review of blockchain's consensus algorithms Comparative Analysis and Future Directions of Blockchain Consensus Mechanisms," *Journal of Computing and Electronic Information Management*, vol. 15, no. 2, pp. 41–45, Dec. 2024, number: 2. [Online]. Available: https://drpress.org/ojs/index.php/jceim/article/view/27804

[36] e. a. N. Katiyar, "Decentralized Consensus Mechanisms in Blockchain: A Comparative Analysis," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 11, no. 11, pp. 706–716, Dec. 2023, number: 11. [Online]. Available: https://ijritcc.org/index.php/ijritcc/article/view/10075

[37] S. Mssassi and A. Abou El Kalam, "The Blockchain Trilemma: A Formal Proof of the Inherent Trade-Offs Among Decentralization, Security, and Scalability," *Applied Sciences*, vol. 15, no. 1, p. 19, Jan. 2025, number: 1 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2076-3417/15/1/19

[38] M. Anus and A. B. Ngadi, "Comparison Analysis of Blockchain Consensus Algorithms in Decentralized Public Environment: A Review," *Asia Proceedings of Social Sciences*, vol. 12, no. 1, pp. 108–112, Mar. 2024. [Online]. Available: https://readersinsight.net/APSS/article/view/3006

[39] "How Many Transactions Per Second - Bitcoin? - Crypto Head," Jun. 2021, section: Cryptocurrency. [Online]. Available: https://cryptohead.io/how-many-transactions-per-second-bitcoin/

[40] K. R. Ramprakash and K. Kunal, "A Study of Blockchain Applications and Challenges in the Contemporary Business World," *Journal of Informatics Education and Research*, vol. 3, no. 2, Nov. 2023, number: 2. [Online]. Available: http://jier.org/index.php/journal/article/view/301

[41] P. Aithal, P. Saavedra, S. Aithal, and S. Ghoash, "Blockchain Technology and its Types-A Short Review," *International Journal of Applied Science and Engineering*, vol. 9, pp. 189–200, Dec. 2021.

[42] X. Chen, S. He, L. Sun, Y. Zheng, and C. Q. Wu, "A Survey of Consortium Blockchain and Its Applications," *Cryptography*, vol. 8, no. 2, p. 12, Jun. 2024, number: 2 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2410-387X/8/2/12

[43] H. Taherdoost, "Smart Contracts in Blockchain Technology: A Critical Review," *Information*, vol. 14, no. 2, p. 117, Feb. 2023, number: 2 Publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2078-2489/14/2/117

[44] J. Singh, S. Rani, and P. Kumar, "Blockchain and Smart Contracts: Evolution, Challenges, and Future Directions," in *2024 International Conference on Knowledge Engineering and Communication Systems (ICKECS)*, vol. 1, Apr. 2024, pp. 1–5. [Online]. Available: https://ieeexplore.ieee.org/document/10616652

[45] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart Contract Development: Challenges and Opportunities," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, Oct. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/8847638

[46] N. Kannengießer, S. Lins, C. Sander, K. Winter, H. Frey, and A. Sunyaev, "Challenges and Common Solutions in Smart Contract Development," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4291–4318, Nov. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9555611

[47] Dr. Vikas Mahandule, Mrs. Harsha Patil, Ms. Pawar Pallavi Dinkar, Ms. Patil Sneha Nitin, Ms. Desai Samiksha Sadashiv, and Ms. Sarowar Komal Subhash, "Blockchain-Enabled Smart Contracts Revolutionizing Supply Chain Management: A Case Study," *International*

Journal of Advanced Research in Science, Communication and Technology, pp. 1–9, Nov. 2024. [Online]. Available: https://ijarsct.co.in/Paper22101.pdf

[48] H. Rapal, "Implementing Smart Contracts in Supply Chain Management | Legitt AI," May 2024. [Online]. Available: https://legittai.com/blog/smart-contracts-in-supply-chain-management

[49] P. Kumar, D. Choubey, O. R. Amosu, and Y. M. Ogunsuji, "Blockchain and smart contracts for supply chain transparency and vendor management," *World Journal of Advanced Research and Reviews*, vol. 23, no. 2, pp. 039–056, 2024, last Modified: 2024-08-05T13:24+05:30 Publisher: World Journal of Advanced Research and Reviews. [Online]. Available: https://wjarr.com/content/blockchain-and-smart-contracts-supply-chain-transparency-and-vendor-management

[50] D. Bellingham, "MediLedger DSCSA Pilot Project."

[51] R. Kamath, "Food Traceability on Blockchain: Walmart's Pork and Mango Pilots with IBM," *The Journal of the British Blockchain Association*, vol. 1, pp. 1–12, Jul. 2018.

[52] "A game changer for global trade." [Online]. Available: https://www.maersk.com/news/articles/2019/09/20/a-game-changer-for-global-trade

[53] port_admin, "Why Maersk and IBM's blockchain platform TradeLens is closing down," Jan. 2023. [Online]. Available: https://piernext.portdebarcelona.cat/en/technology/the-closure-of-tradelens/

[54] N. Khan, S. Hossain, U. Khadka, and S. Sarkar, "Blockchain in Supply Chain Management: Enhancing Transparency, Efficiency, and Trust," *AIJMR - Advanced International Journal of Multidisciplinary Research*, vol. 2, no. 5, Oct. 2024, publisher: AIJMR. [Online]. Available: https://www.aijmr.com/research-paper.php?id=1105

[55] A. Das, J. Hoffmann, and F. Pfenning, "Nomos: A Protocol-Enforcing, Asset-Tracking, and Gas-Aware Language for Smart Contracts," 2016.

[56] Y. Tong, W. Tan, J. Guo, B. Shen, P. Qin, and S. Zhuo, "Smart Contract Generation Assisted by AI-Based Word Segmentation," *Applied Sciences*, vol. 12, no. 9, p. 4773,

Jan. 2022, publisher: Multidisciplinary Digital Publishing Institute. [Online]. Available: https://www.mdpi.com/2076-3417/12/9/4773

[57] P. Qin, W. Tan, J. Guo, and B. Shen, "Intelligible Description Language Contract (IDLC) – A Novel Smart Contract Model," *Information Systems Frontiers*, vol. 26, pp. 1597–1614, May 2021.

[58] O. Choudhury, N. Rudolph, I. Sylla, N. Fairoza, and A. Das, "Auto-Generation of Smart Contracts from Domain-Specific Ontologies and Semantic Rules," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, Jul. 2018, pp. 963–970. [Online]. Available: https://ieeexplore.ieee.org/document/8726491

[59] M. Aparício, S. Guerreiro, and P. Sousa, "Automated demo action model implementation using blockchain smart contracts: 12th International Conference on Knowledge Discovery and Information Retrieval, KDIR 2020 - Part of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2020," *KEOD*, pp. 283–290, 2020, publisher: SciTePress. [Online]. Available: https://www.scopus.com/pages/publications/85107278468

[60] M. Aparıcio, S. Guerreiro, and P. Sousa, "Decentralized Enforcement of DEMO Action Rules Using Blockchain Smart Contracts," 2021.

[61] E. Tallyn, J. Revans, E. Morgan, K. Fisken, "(PDF) Enacting the Last Mile: Experiences of Smart Contracts in Courier Deliveries," in *ResearchGate*. [Online]. Available: https://www.researchgate.net/publication/351422127_Enacting_the_Last_Mile_Experiences_of_Smart_Contracts_in_Courier_Deliveries

[62] D. Aveiro and J. Oliveira, "Towards DEMO Model-Based Automatic Generation of Smart Contracts," in *Advances in Enterprise Engineering XVI*, C. Griffo, S. Guerreiro, and M. E. Iacob, Eds. Cham: Springer Nature Switzerland, 2023, pp. 71–89.

[63] D. Pacheco, D. Aveiro, D. Pinto, and B. Gouveia, *Towards the X-Theory: An Evaluation of the Perceived Quality and Functionality of DEMO's Process Model*, Jul. 2022.

[64] D. Pacheco, D. Aveiro, B. Gouveia, and D. Pinto, "Evaluation of the Perceived Quality and Functionality of Fact Model Diagrams in DEMO," in *Advances in Enterprise Engineering XV*, D. Aveiro, H. A. Proper, S. Guerreiro, and M. de Vries, Eds. Cham: Springer International Publishing, 2022, pp. 114–128.

[65] D. Pinto, D. Aveiro, D. Pacheco, B. Gouveia, and D. Gouveia, "Validation of DEMO's Conciseness Quality and Proposal of Improvements to the Process Model," in *Advances in Enterprise Engineering XIV*, D. Aveiro, G. Guizzardi, R. Pergl, and H. A. Proper, Eds. Cham: Springer International Publishing, 2021, pp. 133–152.

[66] B. Gouveia, D. Aveiro, D. Pacheco, D. Pinto, and D. Gouveia, "Fact Model in DEMO - Urban Law Case and Proposal of Representation Improvements," in *Advances in Enterprise Engineering XIV*, D. Aveiro, G. Guizzardi, R. Pergl, and H. A. Proper, Eds. Cham: Springer International Publishing, 2021, pp. 173–190.

[67] V. Freitas, D. Pinto, V. Caires, L. Tadeu, and D. Aveiro, "The DISME low-code platform - from simple diagram creation to system execution."

[68] J. Dietz and H. Mulder, *Enterprise Ontology: A Human-Centric Approach to Understanding the Essence of Organisation*, Jan. 2020.

[69] "A Blockchain Platform for the Enterprise — Hyperledger Fabric Docs main documentation." [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/

[70] "Quorum." [Online]. Available: https://goquorum.readthedocs.io/

[71] "MultiChain | Enterprise blockchain platform." [Online]. Available: https://www.multichain.com/

[72] "Corda." [Online]. Available: https://r3.com/r3-labs/corda/

[73] "Ethereum.org- o guia completo da Ethereum." [Online]. Available: https://ethereum.org/pt/

[74] "Hello future." [Online]. Available: https://hedera.com/

[75] "Hyperledger." [Online]. Available: https://github.com/hyperledger

[76] L. Abreu, "DEMO Model-Driven Automatic Smart Contract Generation using Hyperledger Fabric," Sep. 2025, publisher: Zenodo. [Online]. Available: https://zenodo.org/records/17178130

[77] V. H. S. Freitas, "Extension of action rule grammar and implementation of processing engine of a DEMO based low-code platform," Jul. 2023. [Online]. Available: http://hdl.handle.net/10400.13/5325

[78] J. L. Dietz and H. Mulder, "The PSI Theory: Understanding the Operation of Organisations | Request PDF," in *ResearchGate*. [Online]. Available: https://www.researchgate.net/publication/340845011_The_PSI_Theory_Understanding_the_Operation_of_Organisations

[79] J. L. Dietz, J. A. Hoogervorst, A. Albani, D. Aveiro, E. Babkin, J. Barjis, A. Caetano, P. Huysmans, J. Iijima, S. van Kervel, H. Mulder, M. Op 't Land, H. A. Proper, J. Sanz, L. Terlouw, J. Tribolet, J. Verelst, and R. Winter, "The discipline of enterprise engineering," *International Journal of Organisational Design and Engineering*, vol. 3, no. 1, pp. 86–114, Jan. 2013, publisher: Inderscience Publishers. [Online]. Available: https://www.inderscienceonline.com/doi/full/10.1504/IJODE.2013.053669

[80] J. L. G. Dietz, "On the Nature of Business Rules," in *Advances in Enterprise Engineering I*, J. L. G. Dietz, A. Albani, and J. Barjis, Eds. Berlin, Heidelberg: Springer, 2008, pp. 1–15.

[81] M. J. G. Andrade, "Modelação e implementação de software de apoio a processos de logística," May 2018. [Online]. Available: http://hdl.handle.net/10400.13/2096

A EE and DEMO

A.1 Design and Engineering Methodology for Organizations (DEMO)

This section is a summary of the DEMO methodology and is reproduced in full from source [77].

A.1.1 Enterprise Engineering

Enterprise Engineering (EE) is an emerging discipline that takes an engineering perspective on enterprises, Fig.19. It aims to address organizational challenges that conventional organizational

sciences are unable to address. Internal organizational transformations are perceived as circumstances necessitating (re)design and (re)implementation [68].

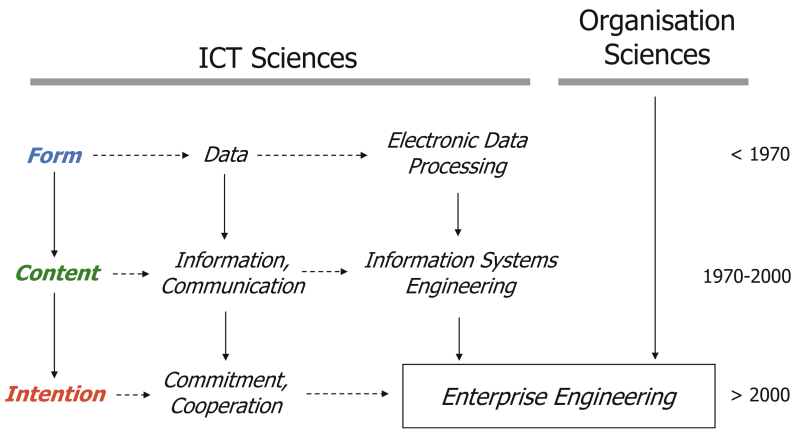


Figure. 19: Enterprise Engineering's roots [78]

EE aims to improve businesses by developing new theories, models, methods, and artifacts for analyzing, designing, implementing, and governing enterprises. It combines management and organization science, information systems science, and computer science [68]. We refer to all structured human endeavors as "enterprises," such as companies, supply chains and others.

The primary objectives of the EE field are intellectual manageability, organizational conciseness, and social responsibility. Intellectual manageability involves understanding complexity and maintaining a perspective on it. Organizational concinnity, is the design, engineering, and implementation of an enterprise to ensure that its operations are cohesive and consistent across all functions. Recognizing that an organization's operations are driven by its operational personnel rather than its managers is the essence of social devotion [79].

A.1.2 DEMO

The approach to an organization's ontology presented subsequently is called DEMO, which is supported by the Ψ-Theory. DEMO is an organization modelling methodology for the analysis and representation of organizational processes through transaction modelling. The PSI (Performance in Social Interaction) theory is about the construction and operation of organizations. The word "organization" indicates that one takes the construction perspective on enterprises. Organizations are systems in the category of social systems, which means that the system elements are social individuals, called actors. The guiding idea is that actors enter into and comply with commitments

towards each other. It explains how and why people cooperate, and in doing so bring about the enterprise's business.

A.1.3 Operation Axiom

According to the operation axiom of the Ψ-Theory [78], subjects in organizations execute two different types of acts: production acts that have an impact on the P-world, or production world, and coordination acts that have an effect on the C-world, or coordination world. Subjects are actors performing an actor role responsible for the execution of these acts. These worlds are always in a particular state, indicated by the C-facts and P-facts that have transpired up to that point in time.

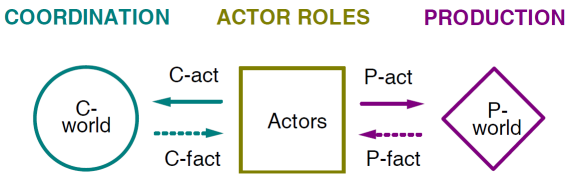


Figure. 20: Interaction of the Actor with the Production and Coordination Worlds [78]

An actor is a subject (human being) in an actor role. The authority and responsibility that the actor may exercise are determined by their role. Coordination acts, which are communication acts, raise commitments. The related coordination fact is produced when a coordination act is completed. For instance, conducting a request act regarding some product results in the fact that the product is requested.

When active, actors consider the status of the P-world and the C-world. Actors continually strive to fulfil the agenda provided by C-facts. In other words, actors engage in interaction through the creation and management of C-facts. Fig. 20 depicts this connection between the actors and the worlds. It illustrates the guiding principle of organizations whose members are dedicated to effectively accomplishing their agenda. The coordination actions are how actors enter into and uphold commitments towards reaching a given production fact. In contrast, the production acts contribute to the organization's objectives by bringing about or delivering products and/or services to the organization's environment [80]

A.1.4 Transaction Axiom

Coordination acts/facts are the atomic building blocks of organisational (but commonly called: business) processes. They always occur in particular patterns of interaction between subjects who play either the initiator role or the executor role in the transaction. These patterns are instances of one generic pattern, called the (business) transaction, in accordance with the transaction axiom of the Ψ-Theory [78]. Examples of process kinds and their respective transaction kinds, in the context of the RAC case, can be seen in Table 5.

Table 5: Process Kinds and associated Transaction Kinds of the RAC case

Process Kind	**Transaction Kind**
Rental	Rental Completing
	Car Taking
	Car Returning
	Deposit Paying
	Invoice Paying
Branch Manage	Branch Creating
Car Manage	Car Creating
Car Type Manage	Car Type Creating
Transport	Transport Managing
	Transport Executing

The terms “initiator” and “executor” replace the colloquial terms “client” and “producer”. In addition, rather than subjects, these terms refer to actor roles. An actor role is described as having the authority and responsibility to carry out a particular type of transaction. Subjects play actor roles in such a way that multiple subjects can play one actor role and one subject can play multiple actor roles. In general, actor roles cannot and should not correspond directly in name or content to organizational positions or functions. As a result, whenever possible, the names of these roles shall be modest variations of the names of the executed transactions. For example, in the RAC scenario, the transaction type “Rental Completing” will be executed by the “rental completer”. An actor role should, in theory, be played by the same subject for all actions that fall under that actor’s purview, namely the initiator’s request and acceptance and the executer’s promise and declaration. By having a precise description of responsibilities, it is possible to have a better and more informed conversation about what the actual requirements for organizational roles or positions should be,

as well as to map actor roles to these roles and, ultimately, to map actual people to the stated roles [81].

Three phases make up the transaction pattern: (1) the order phase, where the initiating actor role of the transaction expresses his wishes in the form of a request and the executing actor role promises to produce the desired result; (2) the execution phase, where the executing actor role produces the desired result; and (3) the result phase, where the executing actor role states the produced result and the initiating actor role accepts that result, effectively closing the transaction. This succession, which can be seen in Fig. 21, is referred to as the "basic transaction pattern", and only takes into account the "happy case", in which everything proceeds as predicted. To realize a new production fact, all five of these steps are essential.

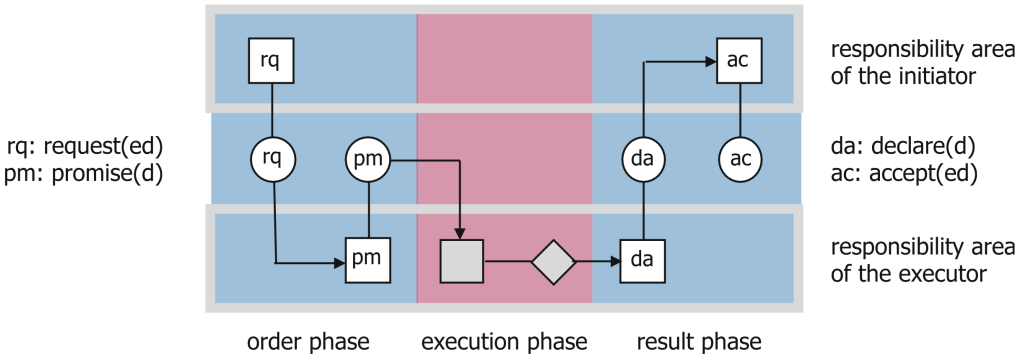


Figure. 21: The basic transaction pattern [68]

Fig. 22 represents an extension of the basic pattern shown above, where the basic transaction process is represented by the green path. The yellow paths depicted here represent cases where the pattern diverges to discussion states (dc and rj). These are represented by a double disk because ending up there is most likely not what the initiator had in mind. Therefore, in the left yellow path's case, the executor is challenged to explain why the request was not accepted, and the initiator gets the chance to counter the executor's arguments and talk about potential adjustments to the product's properties. In the RAC case, the rental completer could have declined the request for a Rental Completing because the provided driver's license isn't valid. In this case, the client performing the request can give up the rental or perform an adapted request with a new submission of a driver's license. This is shown in Fig. 22 by the left yellow path. The executor can then perform the promise of the adapted request, thus following the basic transaction pattern in the green path. It should be noted that the cardinality range 0..1 indicates that executing a new request is optional.

The procedure continues in the declined state, and the process is possibly cancelled if the initiator chooses to do nothing in its place. The initiator may also reject the declare state, which brings the transaction process to the rejected state, where the logic discussed for the left-side yellow path is applied in the same way.

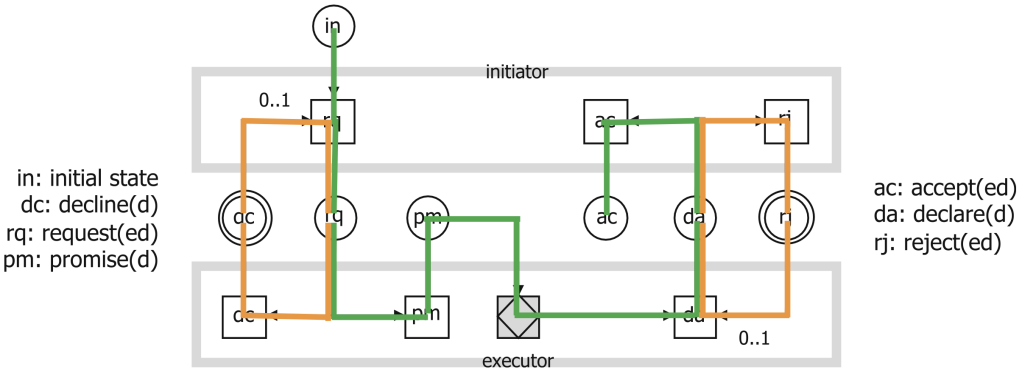


Figure. 22: The standard transaction pattern [68]

The second extension of the basic transaction pattern consists in adding four revoke patterns, one for each of the main steps of the transaction (request, promise, declaration, and acceptance). In other words, from any state within the main transaction process, both the initiator and the executor can undo any fundamental step they have already taken. They can also repeatedly revoke a step inside a single transaction. Revoking a step indicates that one wants to reverse a deliberate action taken previously because one has changed one's mind. It is possible to cancel a mistaken action (as long as the addressee has not responded).

For a better understanding, let's resort to the RAC case to explain the process of revoking a request act, that can be seen in Fig. 23. After submitting a request for a rental completing containing a car of the economic type, the renter realizes that a convertible car is a better choice because of the weather. Therefore, he expresses his wish to the rental completer. He will then evaluate the renter's desire, which will result in the rental completer refusing (yellow path) or allowing this revocation of the request (blue path). If it is allowed, a new request will be submitted with the updated car type and the process will continue (green and white path). If refused, the transaction will move to the refused state and the process will be possibly cancelled. Similar thoughts can be conceived for the revoking of the other three main steps of a transaction.

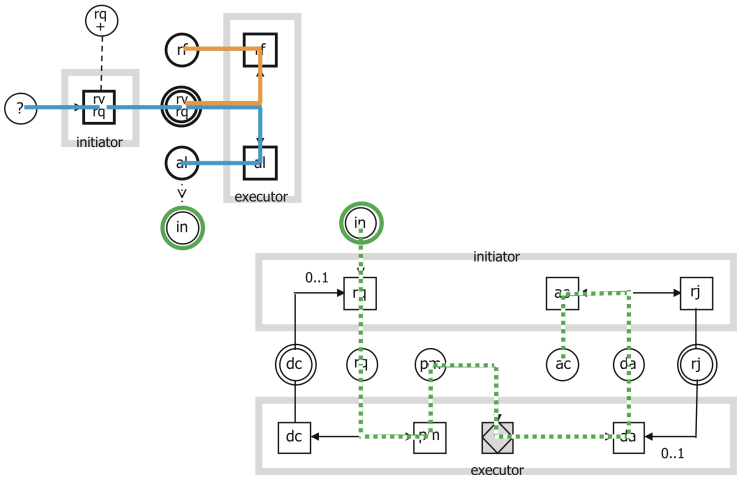


Figure. 23: The process of revoking a request act [68]

The basic transaction pattern with these two extensions results in the complete transaction pattern, shown in Fig. 24. This pattern is considered to be a socio-economic law: all transactions in every type of organization follow paths of acts along this pattern.

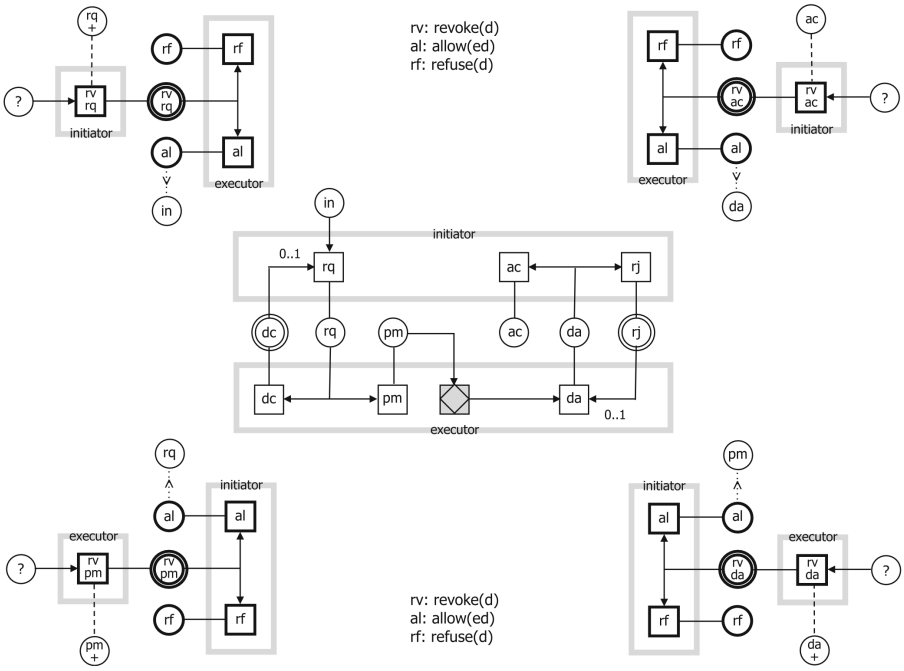


Figure. 24: The complete transaction pattern [68]

Every transaction (instance) is of a particular transaction kind. A transaction kind concerns one specific product kind and has one specific actor role as its executer role. All transactions go through the four social commitment coordination acts of request, promise, state, and accept;

however, these steps might be taken tacitly, that is, without any kind of explicit communication taking place. Particularly, nodding or similar non-verbal behavior is frequently used to realize the promise and acceptance. This could occur as a result of the adage "no news is good news" or just plain forgetfulness, both of which can seriously damage a business. In situations of failure, this becomes more important. All coordinating actions should be carried out explicitly to minimize misunderstandings. Therefore, it's crucial to always take the complete transaction pattern into account while designing organizations. The responsible actor may also not carry out these steps because the relevant subjects may delegate one or more of the transaction steps that fall under their purview to another subject, even if they are still ultimately liable for such acts [67, 80, 81].

The abstraction made by the Ψ-theory to arrive at the ontological model of an organization is the application of the transaction axiom, resulting in a huge reduction in complexity of about 70%, at the documentation level [67, 81].

A.1.5 Distinction Axiom

To fully understand an organization's operation essence, one must understand the three human abilities that are distinguished in performing coordination acts: forma, informa, and performa. This distinction gives rise to three levels of correspondence in the communication between subjects: the forma level (notational correspondence), the informa level (cognitive correspondence), and the performa level (social correspondence), as shown in Fig. 25. To be successful, all three conditions of correspondence must be satisfied, that is, the communication must be free of distortion. The medium level, which is below the forma level, is where forms are encoded in tangible materials and transferred between people. Although equally necessary for effective communication, this level is seen as being outside the purview of EE.

As can be seen in Fig. 26, in the forma level, we have the ability to act at the formative level of coordination, like uttering (speaking, writing) and perceiving (listening, reading) sentences, as well as to perform documental production, like storing, retrieving, transmitting, and copying sentences. The ability to act at the informative level of coordination, like expressing facts (formulating) and educing facts (interpreting), as well as to perform informational production, like remembering and recalling facts, and computing new facts from existing ones refers to the informa level. The final upper performa level is where one has the ability to act at the performative level of coordination, like exposing commitment (by the performer) and evoking commitment (by the addressee), as well

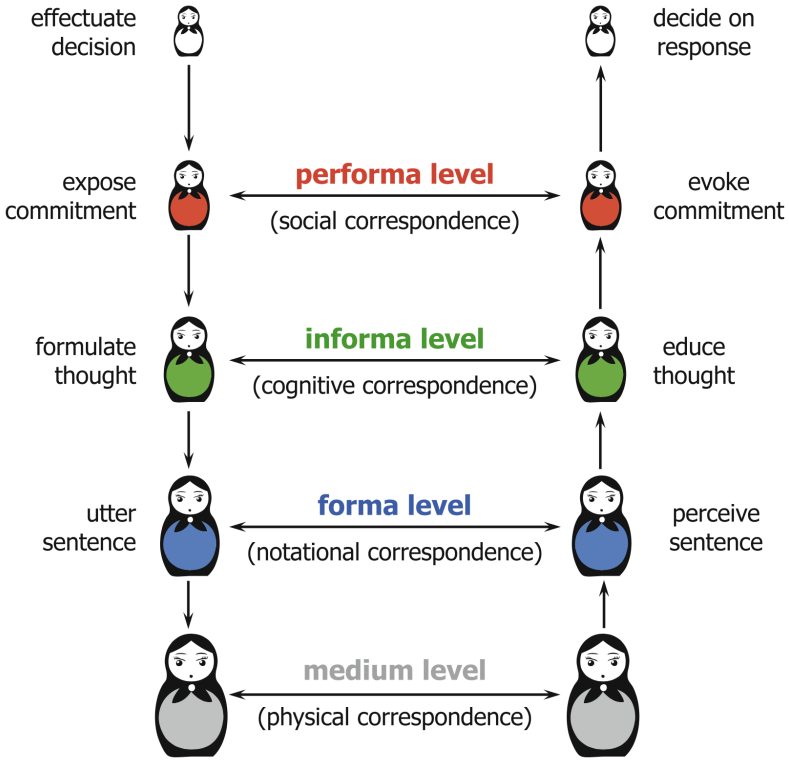


Figure. 25: The process of a communicative act [68]

as performing original production, i.e. creating new facts by deciding, judging, manufacturing or observing things in the physical world.

When it comes to production, the forma ability concerns the documental or datalogical production (store, copy), the informa ability involves the informational or infological production (deduce, compute, calculate) and the performa human ability concerns the essential production in an organization (create, decide, judge), therefore also called ontological production [67, 81].

The abstraction made in the Ψ-theory with the axiom of distinction makes it so that to arrive at the ontological model of an organization only the performa skill with respect to production is considered, thus abstracting from production acts at the datalogical and infological level. This results in a second major reduction in complexity, estimated also at about 70% in terms of documentation [67, 81].

A.1.6 DEMO Ontological Modeling

Like every proper methodology, DEMO comprises a Way of Thinking (WoT), a Way of Modelling (WoM), and a Way of Working (WoW). The WoT consists of a couple theories, with the main

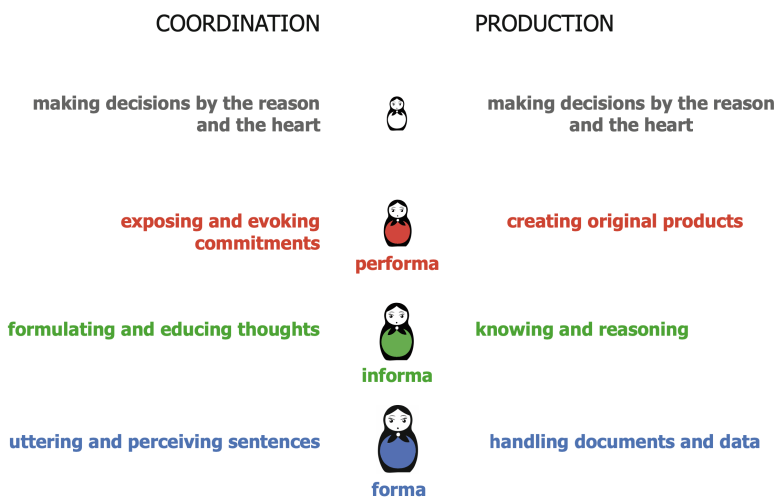


Figure. 26: Human abilities in coordination and production [68]

one being the Ψ-Theory. The WoM consists of an integrated whole of four aspect models: the Cooperation Model (CM), the Action Model (AM), the Process Model (PM), and the Fact Model (FM). These models, depicted in Fig. 27, constitute the complete ontological model of the business organization and subsequently represent the corresponding entity's ontological model. Although a brief explanation is given below for each one, we will be focusing mainly on the Action Model, since it is the one that this project has a direct impact on.

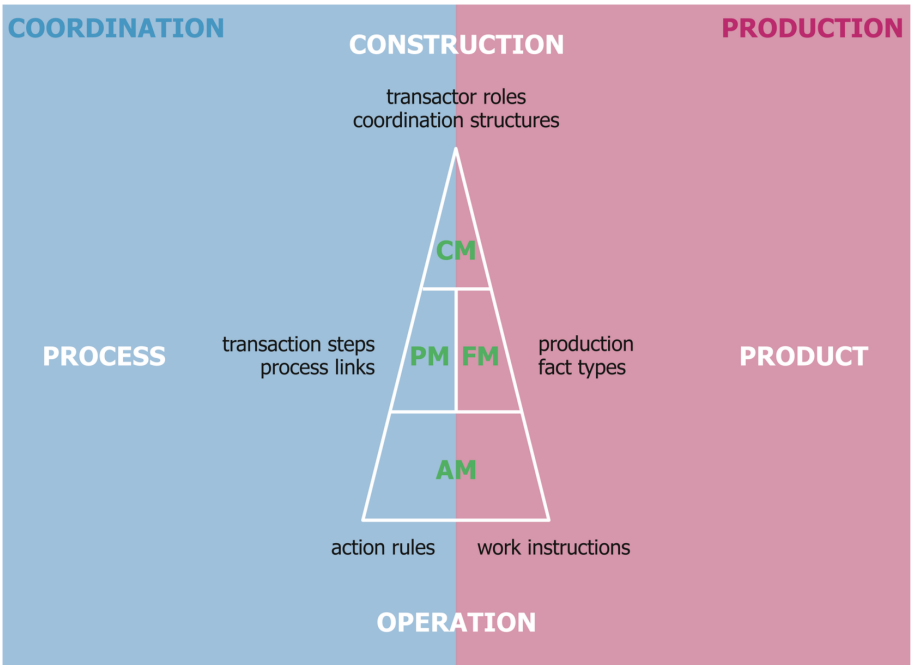


Figure. 27: The integrated DEMO aspect models [68]

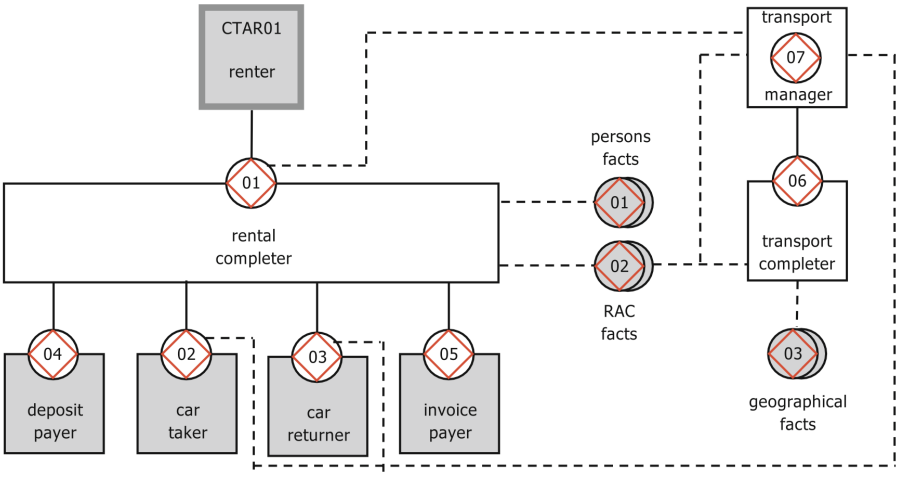
Every time we refer to an organization, a specific Scope of Interest (SoI) is intended. An SoI may include (a portion of) a business or (a portion of) a network of businesses.

The CM of an organization is the ontological model of its construction, thus of the identified transactor roles and the coordination structures among them. A CM is expressed in a Coordination Structure Diagram (CSD) and a Transaction Product Table (TPT), possibly supplemented by a Bank Contents Table and a Bank Access Table. The CM of the RAC case is depicted in Fig. 28 and Table 6. To enhance comprehension, Fig. 28b presents the legend corresponding to the CSD represented in Fig. 28a. Within Table 6, the abbreviations TK, PK, and AR stand for Transaction Kind, Product Kind, and Actor Role, respectively.

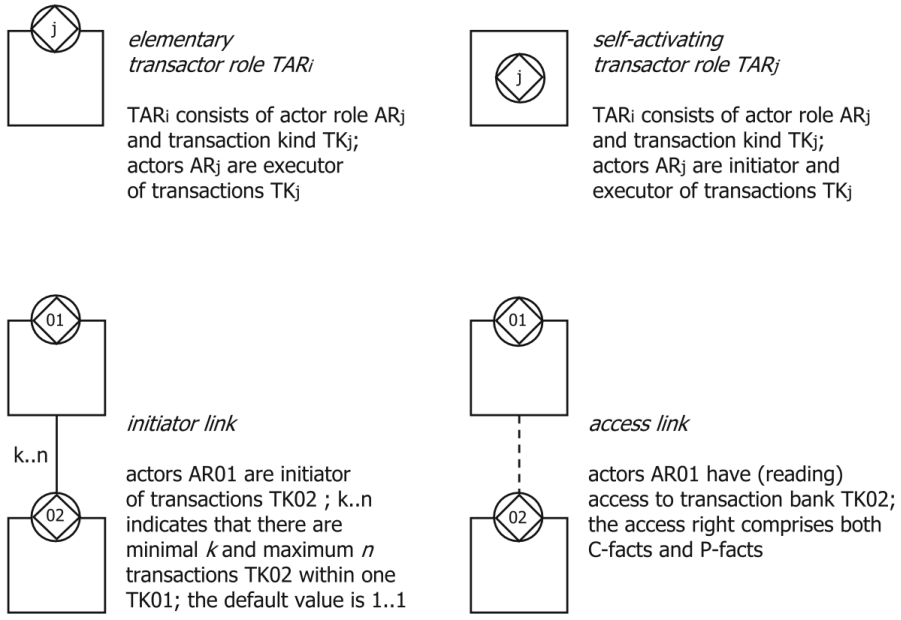
Table 6: Transaction Product Table of the Rental Process in the RAC case [78]

Transaction Kind	**Product Kind**	**Executer Role**
TK01 Rental Completing	**PK01** [rental] **is** completed	**AR01** rental completer
TK02 Car Taking	**PK02** **the** car **of** [rental] **is** taken	**AR02** car taker
TK03 Car Returning	**PK03** **the** car **of** [rental] **is** returned	**AR03** car returner
TK04 Deposit Paying	**PK04** **the** deposit **of** [rental] **is** paid	**AR04** deposit payer
TK05 Invoice Paying	**PK05** **the** invoice **of** [rental] **is** paid	**AR05** invoice payer
TK06 Transport Executing	**PK06** [transport] is executed	**AR06** transport executer
TK07 Transport Managing	**PK07** transport managing **for** [day] **is** done	**AR07** transport manager

The PM of an organization is the ontological model of the state space and the transition space of its coordination world. It contains the existence laws and occurrence laws for all internal and border transactor roles. The PM connects the CM and the AM of an SoI as far as coordination is concerned. A PM is expressed in a Process Structure Diagram, optionally supplemented by a number of Transaction Process Diagrams and a Create Use Table. The Process Model of the Rental process in the RAC case can be seen in Fig. 29. To enhance comprehension, Fig. 29b presents the legend corresponding to the CSD represented in Fig. 29a. The wait link between (T04/ac) and [T02/rq] is a policy requirement set by Rent-A-Car. It specifies that the rental car can only be taken after the deposit has been paid. The wait links between (T02/ac) and [T03/rq], as well as between (T03/ac) and [T05/rq], are necessary for logistical reasons. The first link states that a car must be taken before it can be returned, while the second waiting link indicates that the rental car must be returned before the final invoice can be prepared and payment can be requested [78].

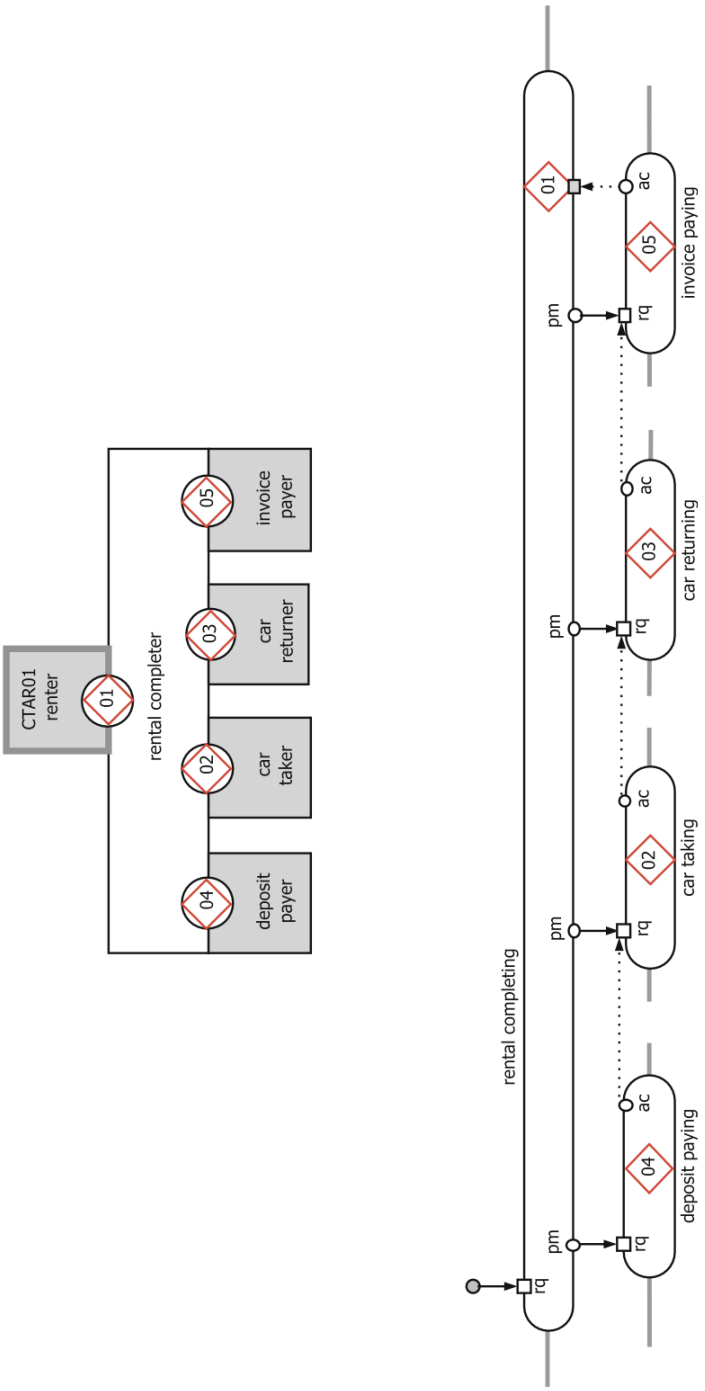


(a) Coordination Structure Diagram of the RAC case

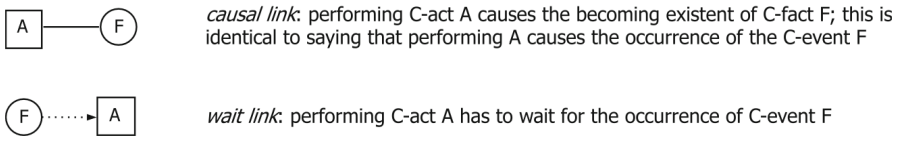


(b) Legend of the Coordination Structure Diagram

Figure. 28: Coordination Structure Diagram of the RAC case [78]



(a) Process Structure Diagram of the Rental Process in the RAC case



(b) Legend of the links in the Process Structure Diagram

Figure. 29: Process Structure Diagram of the Rental Process in the RAC case [78]

The FM of an organization is the ontological model of the state space and the transition space of its production world. It contains the existence laws and occurrence laws for all identified entity types, value types, property types, attribute types, and event types. An FM is expressed in an Object Fact Diagram, supplemented by Derived Fact Specifications, and optionally supplemented by Existence Law Specifications.

B EBNF grammar

B.1 EBNF grammar specification

In the realm of computer science, Extended Backus–Naur Form (EBNF) constitutes a spectrum of meta-syntax languages, each variant capable of representing a context-free grammar. The primary application of EBNF lies in its utility for the formal articulation of formal languages, inclusive of programming languages utilized in computing.

The subsequent section delineates tables that elucidate the syntax employed in the formulation of Action Rules. The table is structured in the following manner: the left column enumerates the concepts requiring definition, where a majority of these concepts correspond to specific blocks in the system, and others relate to selectable fields within these blocks. Conversely, the right column of the table is dedicated to providing the explicit definitions of these concepts.

Additionally, this section introduces new components of our EBNF-based system. The complete grammar, along with further details of these new components, will be presented in the appendices, providing a comprehensive overview of the system's syntax and its extensions.

Before analyzing the upcoming tables that illustrate the grammar implemented by Extended Backus–Naur Form (EBNF), it is important to acknowledge keynotes on this grammar:

– "( )" means grouping

– "[ ]" is for optional elements (zero or one time)

– " " is for optional elements (zero or more times)

– "|" means alternative

– "-" means Syntactic Exception. Separates a rule which must be used (on the left) from the rule describing what is not allowed to be used (on the right) ("Consonant = Letter - Vowel;").

If there's nothing on the right, it can be interpreted only as something that must be used ("OneOrMore = Something-;").

– elements with UPPERCASE mean a terminal symbol with specific behavior assigned to the system/dashboard.

Table 7: DISME's Complete Action Rule's EBNF Syntax

when	WHEN transaction_type IS | **HAS_BEEN** transaction_state { action }-
transaction_type	STRING NOTE: has to be a transaction_type specified in the system.
transaction_state	INITIAL | REQUESTED | PROMISED | EXECUTED | DECLARED | ACCEPTED | DECLINED | REJECTED | STOP | QUIT | REVOKE_REQUEST_REQUESTED | REVOKE_REQUEST_ALLOWED | REVOKE_REQUEST_REFUSED | REVOKE_PROMISE_REQUESTED | REVOKE_PROMISE_ALLOWED | REVOKE_PROMISE_REFUSED | REVOKE_DECLARE_REQUESTED | REVOKE_DECLARE_ALLOWED | REVOKE_DECLARE_REFUSED | REVOKE_ACCEPTANCE_REQUESTED | REVOKE_ACCEPTANCE_ALLOWED
action	causal_link | assign_expression | user_input | **edit_entity_instance** | user_output | produce_doc | if | while | foreach | *API_CALL*
*user_* output	STRING NOTE: special HTML code defined in the template manager component, dashboard will output this to the user interface.
produce_doc	static_template | form_template
static_template	STRING NOTE: special rendered HTML code annotated with custom directives from which a PDF is generated.
form_template	STRING NOTE: special rendered HTML code annotated with custom directives from which a PDF is generated. The interpretation of some directives prompts the end-user at runtime for additional input (e.g. an observation to be placed in the PDF).
assign_expression	property "=" term | property_value
property	STRING NOTE: has to be an existent property specified in the system.

causal_ link	transaction_type MUST BE transaction_state [min [max]] **[CANCEL_PROC] [CONTINUE_IF_SAME_USER]** NOTE: min max are optional and by default come with 1 as prefilled; if min doesn't exist, by default = 1. If max doesn't exist, by default = min. Cancel_proc refers to whether the causal_link cancels the current process. Continue_if_same_user marks whether the execution engine should take the user directly to the execution of the causal_link task when it reaches this action, in case the user is an executor of the created task.
min	Integer
max	Integer | *
user_ input	{ form_property }-
edit_entity_instance	{ entity_detail } { form_property | form_ent_type }- NOTE: properties inserted here must have the flag 'editable' as true.
form_property	property [form_calculation] [enable_condition] { validation_condition }- [MANDATORY] NOTE: mandatory is a checkbox in the block that specifies if the filling of the property should be mandatory in its respective form.
form_ent_type	LINKING_PROPERTY { form_property }- NOTE: one_to_many or many_to_many ent_types only. Linking_property refers to which property is assigned automatically by the engine (ex.: property Car in Car has Feature'). Form_property will be all of the remaining properties from this ent_type.
entity_detail	property NOTE: properties specified here are to be shown in the entity selection modal select box in 'edit entity instance' actions.
form_calculation	compute_expression NOTE: used when we want that the value assigned to a property is automatically computed based on some expression which will fetch data either from values in properties on the current form itself, or properties from the database (currently only supports properties included in the current form).
enable_condition	ENABLE condition NOTE: this is used when we want that a property is "hidden/disabled" from the form unless the specified condition is true, which in that case the property will be shown.
validation_ condition	[NOT] validation_condition_type [EXTRA_FIELD_1] [EXTRA_FIELD_2] [user_output] NOTE: The not, extrafield1 and extrafield2 fields will appear depending on the validation_condition_type chosen. EXTRA_FIELD_1 and EXTRA_FIELD_2 examples are the min and max fields if the validation_condition_type is "Belongs Range".

validation_condition_type	REQUIRED | IS_NUMBER | IS_INTEGER | EQUAL_TO | MAX_WORD_LENGTH | LESS_EQUAL | HIGHER_EQUAL | HIGHER_THAN | LESS_THAN | MIN_LENGTH | BELONGS_RANGE | MAX_LENGTH | MIN_WORD_LENGTH | MAX_WORD_LENGTH | HAS_CHARACTER | HAS_WORD | IS_EMAIL | IS_URL | CUSTOM_VALIDATION | REG_EXPRESSION
if	IF condition THEN { action }- [ELSE { action }-]
condition	(ISTRUE | NOT evaluated_expression | condition) | (AND | OR { evaluated_expression | condition }-) NOTE: if condition is of type ISTRUE engine will evaluate the expression; if condition is of type NOT, engine will evaluate either the expression or condition; if it is of type AND or OR it will accordingly evaluate the specified expressions/conditions
evaluated_expression	comp_evaluated_expression | user_evaluated_expression
comp_evaluated_expression	term logical_operator term | property_value
user_evaluated_expression	STRING NOTE: this is a simple text input and dashboard shows this "textual informal expression" that has to be evaluated by the user who will decide on a result of true or false.
logical_ operator	"<" | ">" | "==" | "!="
property_value	STRING NOTE: must be a possible value of a property with 2 cases: 1) value_type is enum and one can select all allowed values for the selected property 2) value_type is prop_ref where possible values to select will be the values associated with the property fk_property_id (e.g., property name of entity type location has many values for the different locations one can pick-up or drop-off rentals).
term	constant | value | property | query | compute_expression | produce_doc
constant	**value_type** STRING NOTE: Can be a constant defined on the system, or can be a new specification of a constant. In the latter case, the name, value and value type of the constant to be created must also be specified.
value	**value_type** STRING NOTE: free value inserted when editing the Action Rule. Must also specify the value type when specifying the new value.
value_ type	*TEXT* | INTEGER_NUMBER | REAL_NUMBER | BOOLEAN | **ENUM** | **DATE** | **TIME**

query	STRING { term } NOTE: has to be an existing query specified in the query table. It will have terms if it's a dynamic query where the user can specify query parameters.
compute_expression	term { compute_operator term }-
compute_operator	"+" | "-" | "×"|"/"|"∧"
while	WHILE condition { action }-
foreach	FOREACH set { action }-
set	SET_OF_ELEMENTS